

3

FPGA Fabrics

Architectures of FPGAs.

SRAM-based FPGAs.

Antifuse-programmed FPGAs.

Programmable I/O pins.

FPGA circuits: logic and interconnect.

3.1 Introduction

In this chapter we will study the basic structures of FPGAs, known as **fabrics**. We will start with a brief introduction to the structure of FPGA fabrics. However, there are several fundamentally different ways to build an FPGA. Therefore, we will discuss combinational logic and interconnect for the two major styles of FPGA: SRAM-based and antifuse-based. The features of I/O pins are fairly similar among these two types of FPGAs, so we will discuss pins at the end of the chapter.

3.2 FPGA Architectures

elements of FPGAs

In general, FPGAs require three major types of elements:

- combinational logic;
- interconnect;
- I/O pins.

These three elements are mixed together to form an FPGA fabric.

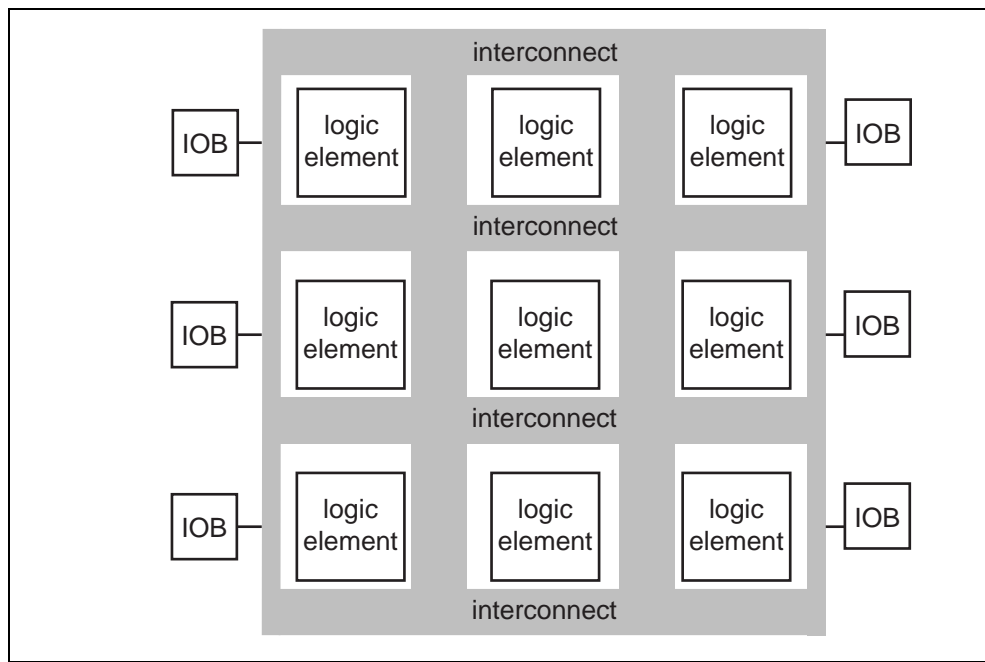


Figure 3-1 Generic structure of an FPGA fabric.

FPGA architectures

Figure 3-1 shows the basic structure of an FPGA that incorporates these three elements. The combinational logic is divided into relatively small units which may be known as **logic elements (LEs)** or **combinational logic blocks (CLBs)**. The LE or CLB can usually form the function of several typical logic gates but it is still small compared to the typical combinational logic block found in a large design. The interconnections are made between the logic elements using programmable interconnect. The interconnect may be logically organized into channels or other units. FPGAs typically offer several types of interconnect depending on the distance between the combinational logic blocks that are to be connected; clock signals are also provided with their own interconnection networks. The I/O pins may be referred to as **I/O blocks (IOBs)**. They are generally programmable to be inputs or outputs and often provide other features such as low-power or high-speed connections.

FPGA interconnect

An FPGA designer must rely on pre-designed wiring, unlike a custom VLSI designer who can design wires as needed to make connections.

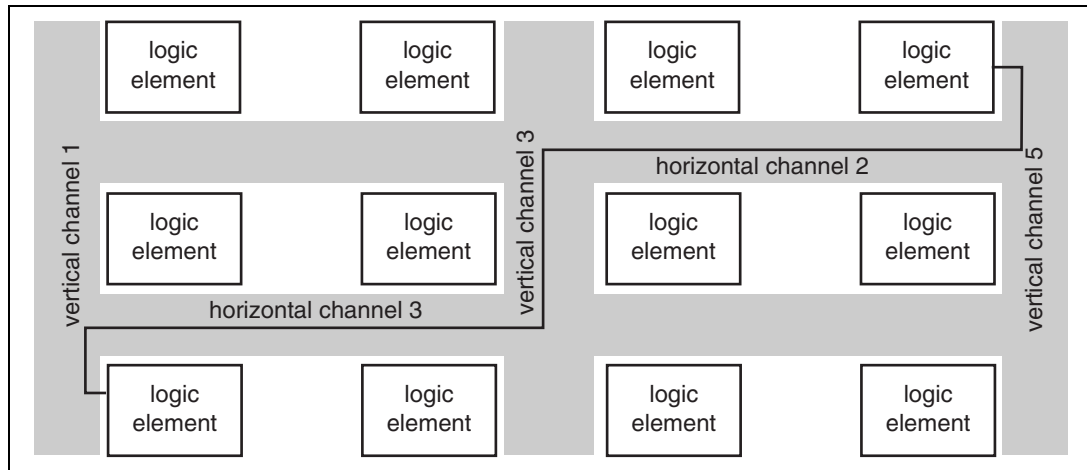


Figure 3-2 Interconnect may require complex paths.

The interconnection system of an FPGA is one of its most complex aspects because wiring is a global property of a logic design.

connection paths

Connections between logic elements may require complex paths since the LEs are arranged in some sort of two-dimensional structure as shown in Figure 3-2. We therefore need to make connections not just between LEs and wires but also between the wires themselves. Wires are typically organized in **wiring channels** or **routing channels** that run horizontally and vertically through the chip. Each channel contains several wires; the human designer or a program chooses which wire will be used in each channel to carry a signal. Connections must be made between wires in order to carry a signal from one point to another. For example, the net in the figure starts from the output of the LE in the upper-right-hand corner, travels down vertical channel 5 until it reaches horizontal channel 2, then moves down vertical channel 3 to horizontal channel 3. It then uses vertical channel 1 to reach the input of the LE at the lower-left-hand corner.

segmented wiring

In order to allow a logic designer to make all the required connections between logic elements, the FPGA channels must provide wires of a variety of lengths, as shown in Figure 3-3. Because the logic elements are organized in a regular array, we can arrange wires going from one LE to another. The figure shows connections of varying length as measured in units of LEs: the top signal of length 1 goes to the next LE, the second signal goes to the second LE, and so on. This organization is

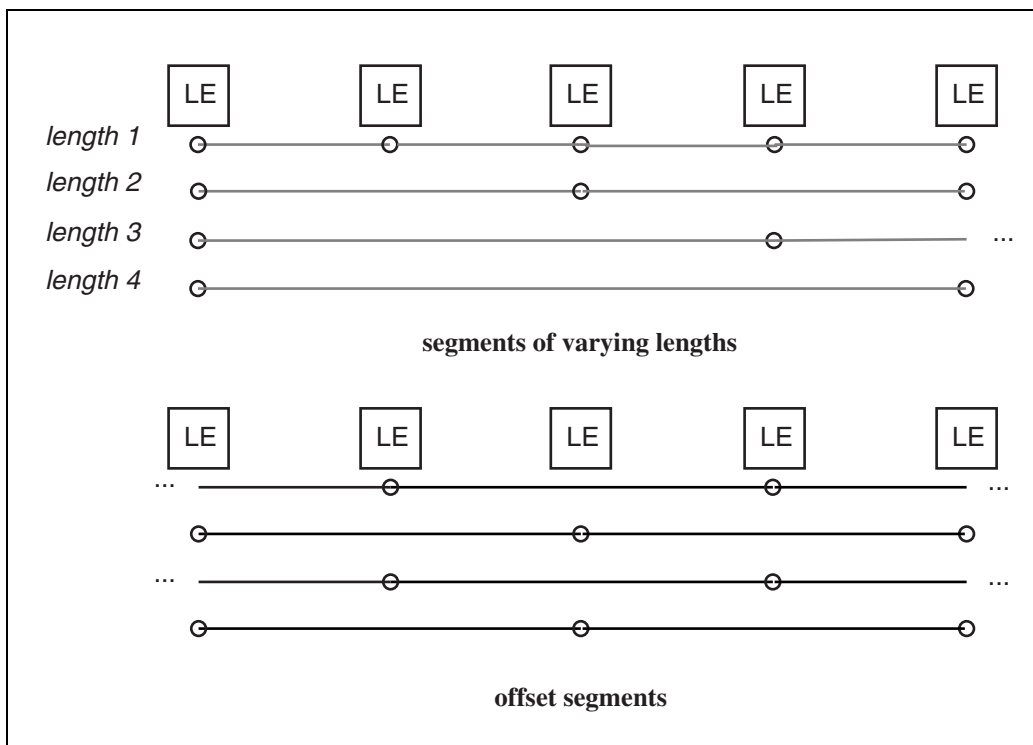


Figure 3-3 Segmented wiring and offsets.

known as a **segmented wiring** structure [ElG88] since the wiring is constructed of segments of varying lengths. The alternative to segmented wiring is to make each wire length 1. However, this would require a long connection to hop through many programmable wiring points, and as we will see in Section 3.6, that would lead to excessive delay along the connection. The segments in a group need not all end at the same point. The bottom part of Figure 3-3 shows segments of length 2 that are offset relative to each other.

FPGA configuration

All FPGAs need to be **programmed** or **configured**. There are three major circuit technologies for configuring an FPGA: SRAM, antifuse, and flash. No matter what circuits are used, all the major elements of the FPGA—the logic, the interconnect, and the I/O pins—need to be configured. The details of these elements vary greatly depending on how the FPGA elements are to be programmed. But FPGAs are very complex VLSI systems that can be characterized in many different ways.

Some of the characteristics of interest to the system designer who wants to use an FPGA include:

- How much logic can I fit into this FPGA?
- How many I/O pins does it have?
- How fast does it run?

While we can determine fairly easily how many I/O pins an FPGA has, determining how much logic can be fit into it and how fast that logic will run is not simple. As we will see in this chapter, the complex architecture of an FPGA means that we must carefully optimize the logic as we fit it into the FPGA. The amount of logic we can fit and how fast that logic runs depends on many characteristics of the FPGA architecture, the logic itself, and the logic design process. We'll look at the tools necessary to configure an FPGA in Chapter 4.

design of FPGA architectures

Some questions of interest to the person who designs the FPGA itself include:

- How many logic elements should the FPGA have?
- How large should each logic element be?
- How much interconnect should it have?
- How many types of interconnection structures should it have?
- How long should each type of interconnect be?
- How many pins should it have?

In Section 3.6 and Section 3.7 we will survey some of the extensive research results in the design of FPGA fabrics. A great deal of theory and experimentation has been developed to determine the parameters for FPGA architectures that best match the characteristics of typical logic that is destined for FPGA implementation.

fine-grain vs. coarse-grain

All of the FPGAs we will deal with in this chapter are **fine-grained** FPGAs—their logic elements can implement fairly small pieces of logic. Advances in VLSI technology are making possible **coarse-grained** FPGAs that are built from large blocks. We will look at some of these architectures in Chapter 7.

chapter outline

The next section looks at FPGAs based on static memory. Section 3.3 studies FPGAs built from permanently programmed parts, either antifuses or flash. Section 3.5 looks at the architecture of chip input and output, which is fairly similar in SRAM and antifuse/flash FPGAs. Section 3.6 builds on these earlier sections by studying in detail the cir-

circuits used to build FPGA elements, and Section 3.7 is a detailed study of the architectural characteristics of FPGAs.

3.3 SRAM-Based FPGAs

Static memory is the most widely used method of configuring FPGAs. In this section we will look at the elements of an FPGA: logic, interconnect, and I/O. In doing so we will consider both general principles and specific commercial FPGAs.

3.3.1 Overview

characteristics of SRAM-based FPGAs

SRAM-based FPGAs hold their configurations in static memory (though as we will see in Section 3.6 they don't use the same circuits as are used in commodity SRAMs). The output of the memory cell is directly connected to another circuit and the state of the memory cell continuously controls the circuit being configured.

Using static memory has several advantages:

- The FPGA can be easily reprogrammed. Because the chips can be reused, and generally reprogrammed without removing them from the circuit, SRAM-based FPGAs are the generally accepted choice for system prototyping.
- The FPGA can be reprogrammed during system operation, providing dynamically reconfigurable systems.
- The circuits used in the FPGA can be fabricated with standard VLSI processes.
- Dynamic RAM, although more dense, needs to be refreshed, which would make the configuration circuitry much more cumbersome.

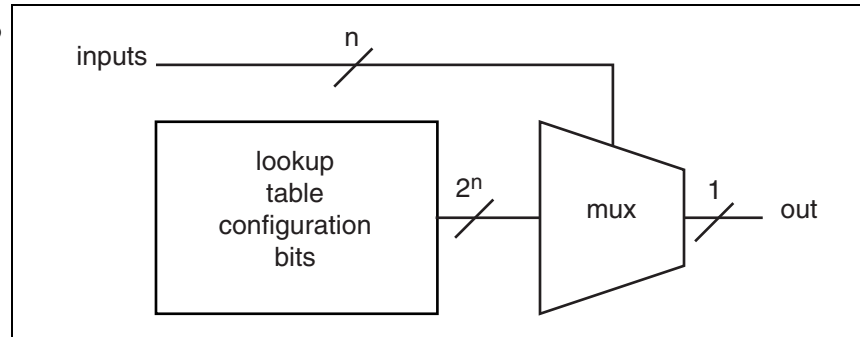
SRAM-based FPGAs also have some disadvantages:

- The SRAM configuration memory burns a noticeable amount of power, even when the program is not changed.
- The bits in the SRAM configuration are susceptible to theft.

A large number of bits must be set in order to program an FPGA. Each combinational logic element requires many programming bits and each programmable interconnection point requires its own bit.

3.3.2 Logic Elements

Figure 3-4 A lookup table.



lookup tables

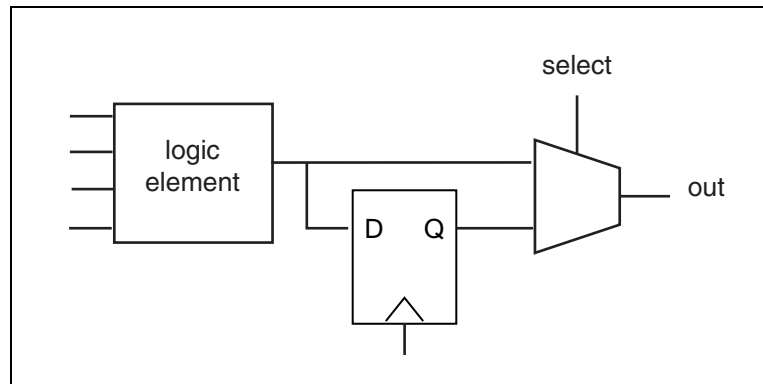
The basic method used to build a **combinational logic block (CLB)**—also called a **logic element** or **LE**—in an SRAM-based FPGA is the **lookup table (LUT)**. As shown in Figure 3-4, the lookup table is an SRAM that is used to implement a truth table. Each address in the SRAM represents a combination of inputs to the logic element. The value stored at that address represents the value of the function for that input combination. An n -input function requires an SRAM with 2^n locations. Because a basic SRAM is not clocked, the lookup table LE operates much as any other logic gate—as its inputs change, its output changes after some delay.

programming a lookup table

Unlike a typical logic gate, the function represented by the LE can be changed by changing the values of the bits stored in the SRAM. As a result, the n -input LE can represent 2^{2^n} functions (though some of these functions are permutations of each other). A typical logic element has four inputs. The delay through the lookup table is independent of the bits stored in the SRAM, so the delay through the logic element is the same for all functions. This means that, for example, a lookup table-based LE will exhibit the same delay for a 4-input XOR and a 4-input NAND. In contrast, a 4-input XOR built with static CMOS logic is considerably slower than a 4-input NAND. Of course, the static logic gate is generally faster than the LE.

Logic elements generally contain registers—flip-flops and latches—as well as combinational logic. A flip-flop or latch is small compared to the combinational logic element (in sharp contrast to the situation in custom VLSI), so it makes sense to add it to the combinational logic element. Using a separate cell for the memory element would simply take up routing resources. As shown in Figure 3-5, the memory element is

Figure 3-5 A flip-flop in a logic element.



connected to the output; whether it stores a given value is controlled by its clock and enable inputs.

complex logic elements

More complex logic blocks are also possible. For example, many logic elements also contain special circuitry for addition.

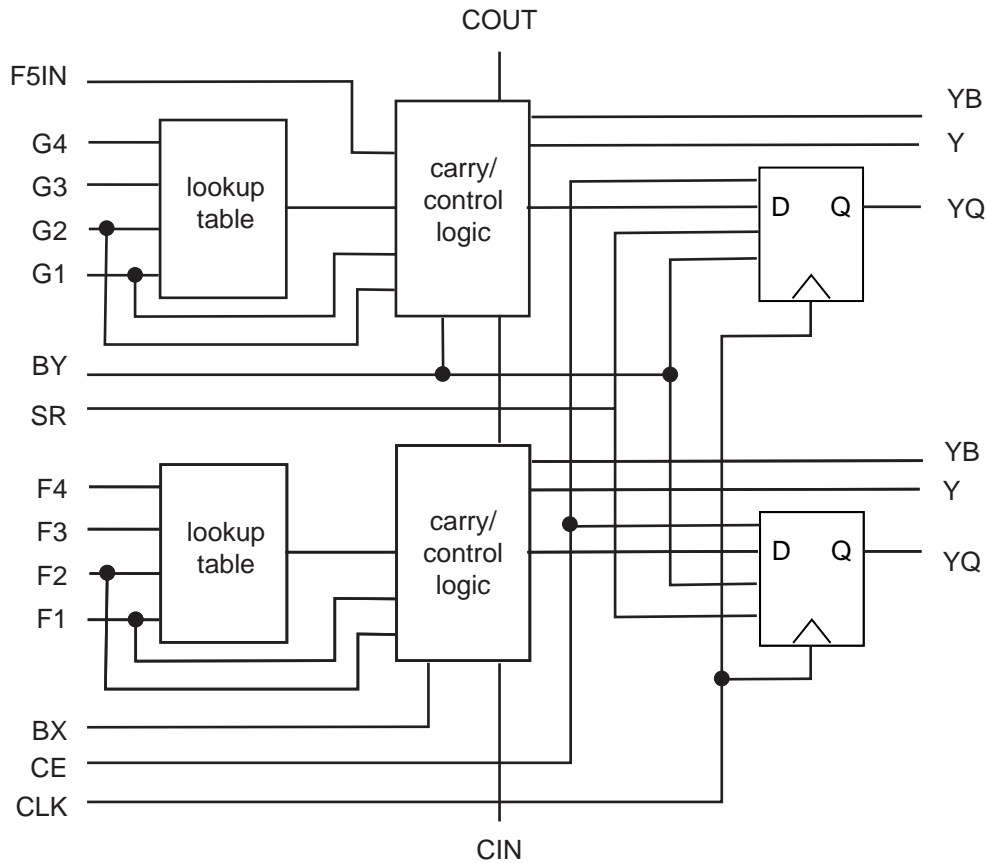
Many FPGAs also incorporate specialized adder logic in the logic element. The critical component of an adder is the carry chain, which can be implemented much more efficiently in specialized logic than it can using standard lookup table techniques.

The next two examples describe the logic elements in two FPGAs. They illustrate both the commonality between FPGA structures and the varying approaches to the design of logic elements.

Example 3-1

Xilinx Spartan-II combinational logic block

The Spartan-II combinational logic block [Xi101] consists of two identical slices, with each slice containing a LUT, some carry logic, and registers. Here is one slice:



A slice includes two **logic cells (LCs)**. The foundation of a logic cell is the pair of four-bit lookup tables. Their inputs are F1-F4 and G1-G4. Each lookup table can also be used as a 16-bit synchronous RAM or as a 16-bit shift register. Each slice also contains carry logic for each LUT so that additions can be performed. A carry in to the slice enters the CIN input, goes through the two bits of carry chain, and out through COUT. The arithmetic logic also includes an XOR gate. To build an adder, the

XOR is used to generate the sum and the LUT is used for the carry computation.

Each slice includes a multiplexer that is used to combine the results of the two function generators in a slice. Another multiplexer combines the outputs of the multiplexers in the two slices, generating a result for the entire CLB.

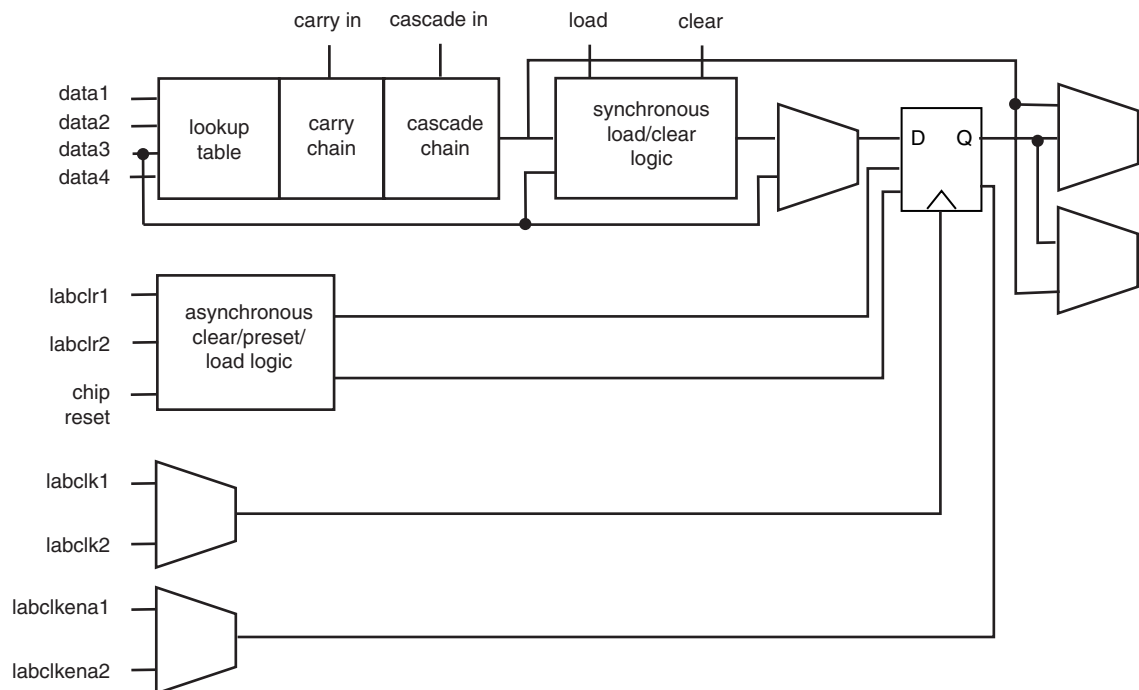
The registers can be configured either as D-type flip-flops or as latches. Each register has clock and clock enable signals.

Each CLB also contains two three-state drivers (known as **BUFTs**) that can be used to drive on-chip busses.

Example 3-2 Altera APEX II logic elements

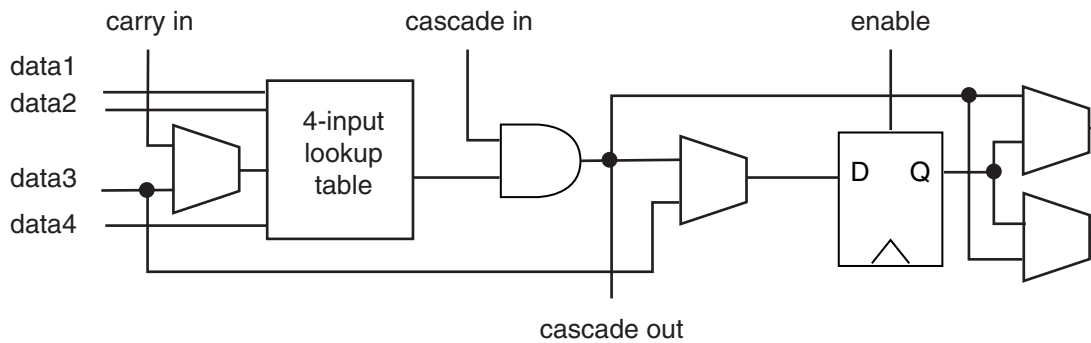
The APEX II's logic [Alt02] is organized into **logic array blocks (LABs)**. Each LAB includes 10 logic elements. Each logic element contains a lookup table, flip-flop, *etc.* The logic elements in an LAB share some logic, such as a carry chain and some control signal generation.

A single logic element looks like this:

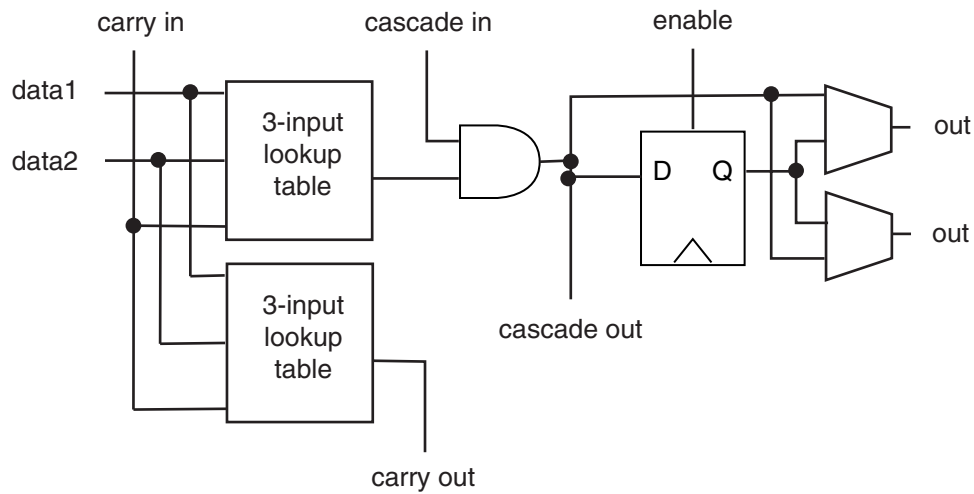


The main logic chain starts with a 4-input lookup table. The output of the LUT is fed to a carry chain. The cascade chain is used for cascading large fanin functions. For example, an AND function with a large number of inputs can be built using the cascade chain. The output of the cascade chain goes to a register. The register can be programmed to operate as a D, T, JK, or SR element.

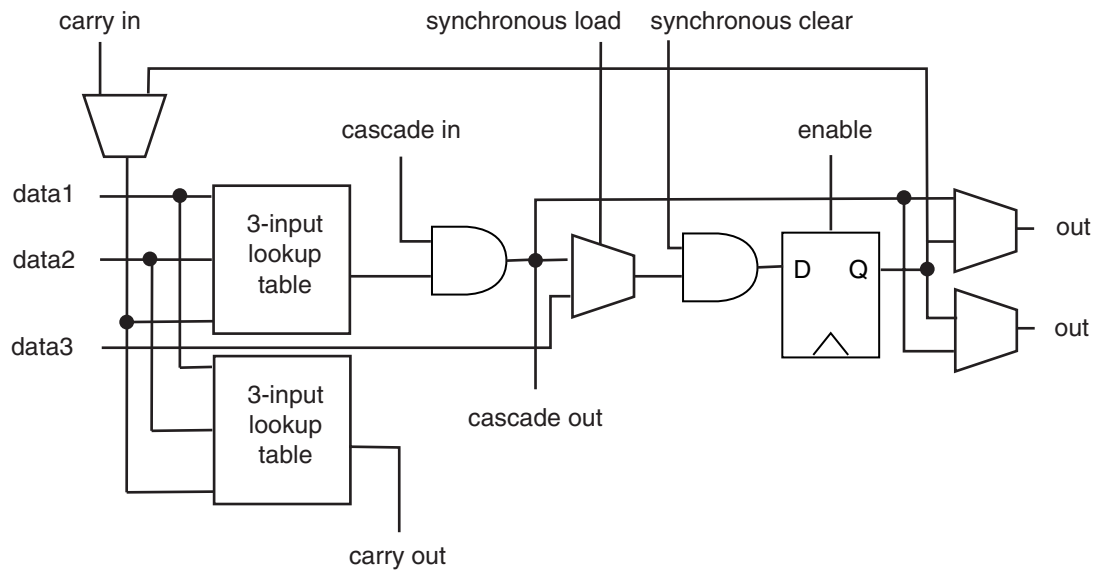
To use all this logic, an LE can be operated in normal, arithmetic, or counter mode. In normal mode, the LE looks like this:



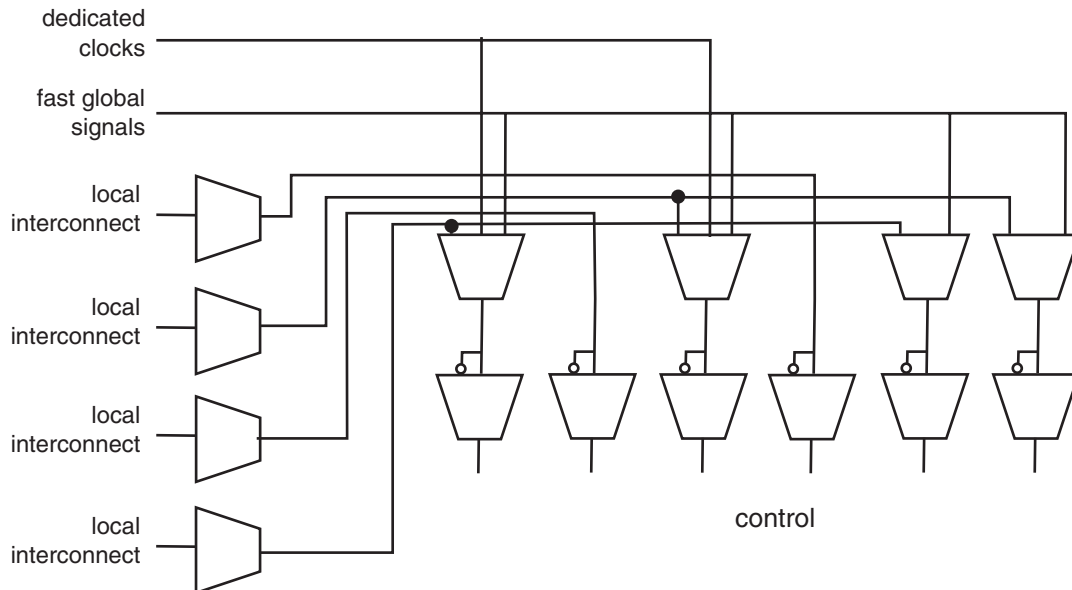
In arithmetic mode, the LE's configuration changes to take advantage of the carry chain logic:



In counter mode, the configuration is altered slightly to provide a fast count:



Each logic array block also includes some logic to generate control signals that can be distributed to all the LEs in the block:



This block can take input signals from the rest of the chip and generate signals for the register: load, clear, enable, *etc.*

The LAB-wide control signals control the preset and clear signals in the LEs' registers. The APEX II also provides a chip-wide reset pin to clear all the registers in the device.

3.3.3 Interconnection Networks

Logic elements must be interconnected to implement complex machines. An SRAM-based FPGA uses SRAM to hold the information used to program the interconnect. As a result, the interconnect can be reconfigured, just as the logic elements can.

*programmable
interconnection points*

Figure 3-6 shows a simple version of an **interconnection point**, often known as a **connection box**. A programmable connection between two wires is made by a CMOS transistor (a pass transistor). The pass transistor's gate is controlled by a static memory program bit (shown here as a

nication. As with high-speed highways with widely spaced exits, they have fewer connection points than local connections. This reduces their impedance. Global wires may also include built-in electrical repeaters to reduce the effects of delay.

- Special wires may be dedicated to distribute clocks or other register control signals.

connections and choice

In order to be able to select routes for connections, the FPGA fabric must provide choices for the interconnections. We must be able to connect a logic element's input or output to one of several different wires. Similarly, the ends of each wire must be able to connect to several different wires. Each of these choices requires its own connection box. This adds up to a large amount of circuitry and wiring that is devoted to programmable interconnection. If we add too many choices, we end up devoting too much of the chip to programmable interconnect and not enough to logic. If we don't have enough choice, we can't make use of the logic resources on the chip. One of the key questions in the design of an FPGA fabric is how rich the programmable interconnect fabric should be. Section 3.7 presents the results of several experiments that are designed to match the richness of FPGA fabrics to the characteristics of typical designs.

types of interconnect

One way to balance interconnect and logic resources is to provide several different types of interconnect. Connections vary both in length and in speed. Most FPGAs offer several different types of wiring so that the type most suited to a particular wire can be chosen for that wire. For example, the carry signal in an adder can be passed from LE to LE by wires that are designed for strictly local interconnections; longer connections may be made in a more general interconnect structure that uses segmented wiring.

The next examples describe the interconnect systems in the two FPGAs we discussed earlier.

Example 3-3 **The Xilinx** **Spartan-II** **interconnect** **system**

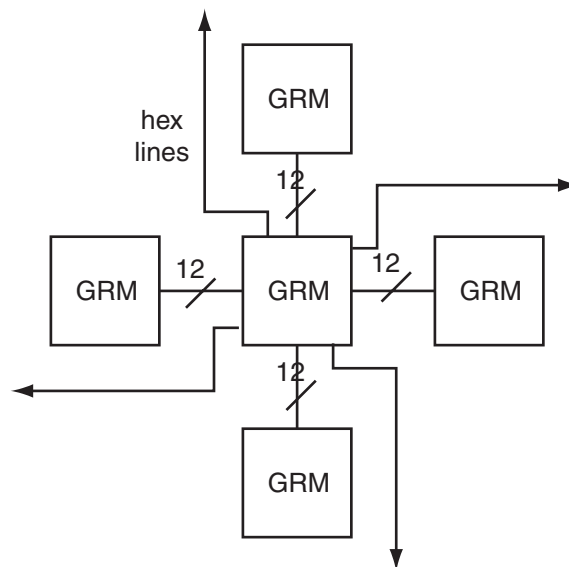
The Spartan-II includes several types of interconnect: local, general-purpose, I/O, global, and clock.

The local interconnect system provides several kinds of connections. It connects the LUTs, flip-flops, and general purpose interconnect. It also provides internal CLB feedback. Finally, it includes some direct paths for high-speed connections between horizontally adjacent CLBs. These paths can be used for arithmetic, shift registers, or other functions that need structured layout and short connections.

The general-purpose routing network provides the bulk of the routing resources. This network includes several types of interconnect:

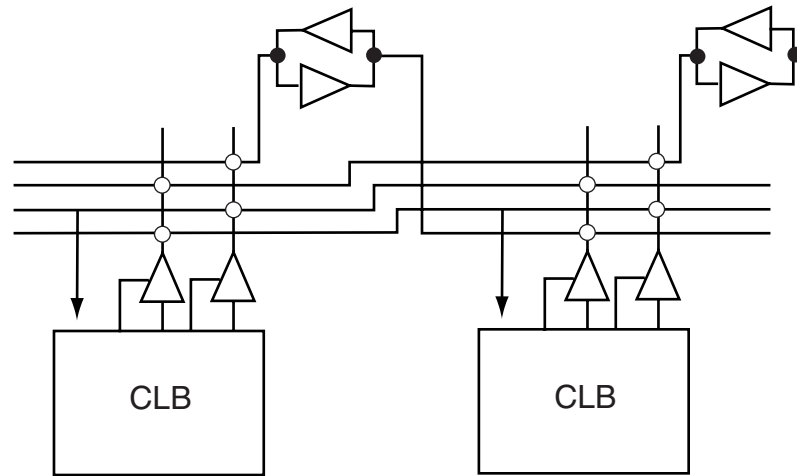
- A **general routing matrix (GRM)** is a switch matrix used to connect horizontal and vertical routing channels as well as the connections between the CLBs and the routing channels.
- There are 24 single-length lines to connect each GRM to the four nearest GRMs, to the left, right, above, and below.
- Hex lines route GRM signals to the GRMs six blocks away. Hex lines provide longer interconnect. The hex lines include buffers to drive the longer wires. There are 96 hex lines, one third bidirectional and the rest unidirectional.
- 12 longlines provide interconnect spanning the entire chip, both vertically and horizontally.

The general routing matrices are related to the single-length and hex lines like this:



Some additional routing resources are placed around the edge of the chip to allow connections to the chip's pins.

One type of dedicated interconnect resource is the on-chip three-state bus:

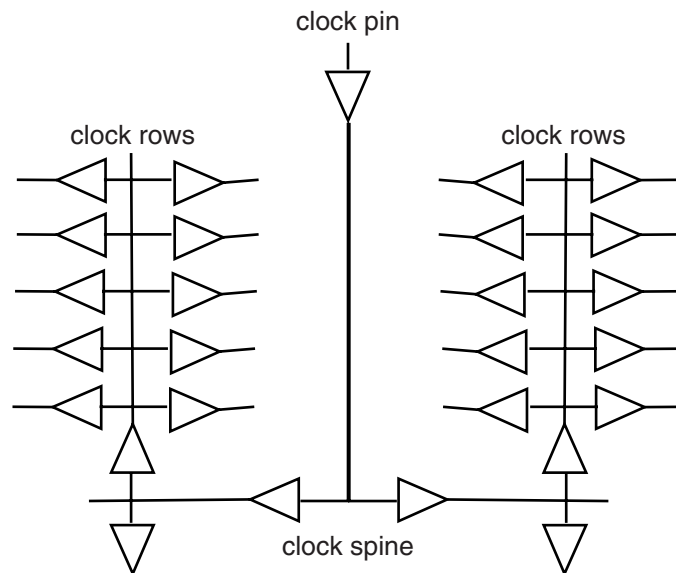


These busses run only horizontally. Four partitionable busses are available per CLB row.

Another type of dedicated routing resource is the wires connecting the carry chain logic in the CLBs.

The global routing system is designed to distribute high-fanout signals, including both clocks and logic signals. The primary global routing network is a set of four dedicated global nets with dedicated input pins. Each global net can drive all CLB, I/O register, and block RAM clock

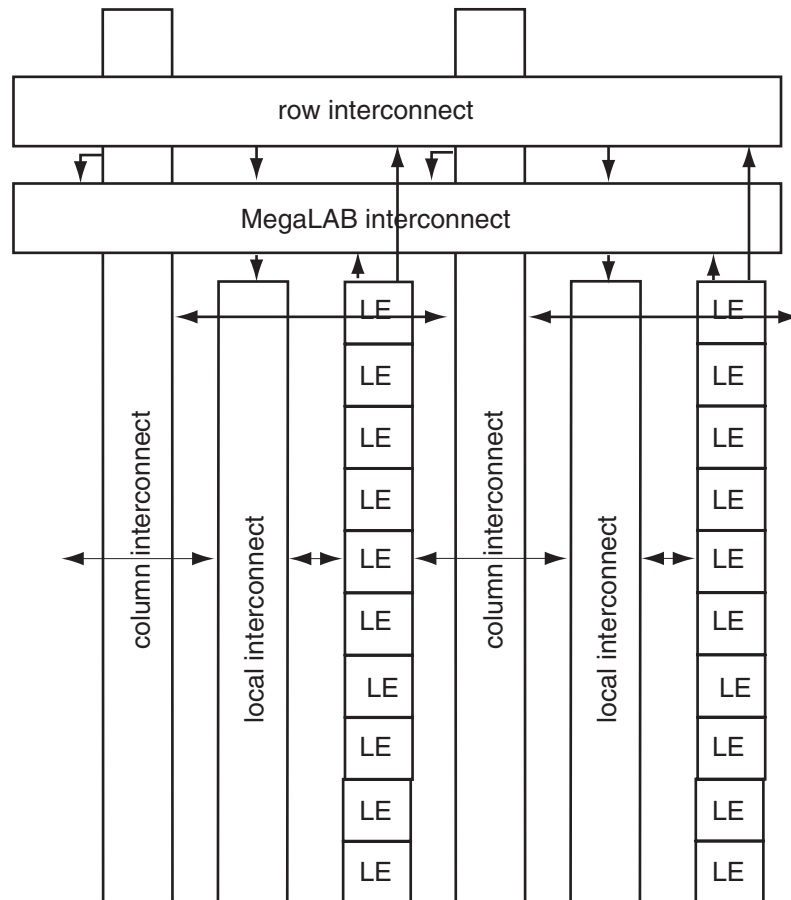
pins. The clock distribution network is buffered to provide low delay and low skew:



The secondary global routing network includes 24 backbone lines, half along the top of the chip and half along the bottom. The chip also includes a delay-locked loop (DLL) to regulate the internal clock.

Example 3-4 The Altera APEX II interconnect system

The APEX II uses horizontal and vertical interconnect channels to interconnect the logic elements and the chip pins. The interconnect structure looks like this:



A row line can be driven directly by an LE, I/O element, or embedded memory in that row. A column line can also drive a row line; columns can be used to connect wires in two rows.

Some dedicated signals with buffers are provided for high-fanout signals such as clocks.

Column I/O pins can directly drive these interconnect lines. Each line traverses two MegaLAB structures, driving the four MegaLABs in the top row and the four MegaLABs in the bottom row of the chip.

3.3.4 Configuration

SRAM configuration

SRAM-based FPGAs are reconfigured by changing the contents of the configuration SRAM. A few pins on the chip are dedicated to configuration; some additional pins may be used for configuration and later released for use as general-purpose I/O pins. Because FPGAs are reconfigured relatively infrequently, configuration lines are usually bit-serial. However, it is possible to send several bits in parallel if configuration time is important.

During prototyping and debugging, we change the configuration frequently. A download cable can be used to download the configuration directly from a PC. When we move the design into production, we do not want to rely on a download cable and a PC. Specialized programmable read-only memories (PROMs) are typically used to store the configuration on the printed circuit board with the FPGA. The FPGA upon power-up runs through a protocol on its configuration pins. The EPROM has a small amount of additional logic to supply a clock signal and answer the FPGA's configuration protocol.

configuration time

When we start up the system, we can usually tolerate some delay to download the configuration into the FPGA. However, there are cases when configuration time is important. This is particularly true when the FPGA will be dynamically reconfigured—reconfigured on-the-fly while the system is operating, such as the Radius monitor described in the next example.

Example 3-5 **Dynamic** **reconfiguration**

The Radius monitors for the Apple Macintosh™ computer [Tri94] operated in horizontal (landscape) and vertical (portrait) modes. When the monitor was rotated from horizontal to vertical or vice versa, the monitor contents changed so that the display contents did not rotate. The Radius monitor used an SRAM-based FPGA to run the display. Because long shift registers to hold the display bits were easily built on the FPGA, the part made sense even without reconfiguration. However, the monitor's mode shift was implemented in part by reconfiguring the

FPGA. A mercury switch sensed the rotation and caused a new personality to be downloaded to the FPGA, implementing the mode switch.

configuration circuits

The configuration memory must be designed to be as immune as possible to power supply noise. Glitches in the power supply voltage can cause memory circuits to change state. Changing the state of a configuration memory cell changes the function of the chip and can even cause electrical problems if two circuits are shorted together. As a result, the memory cells used in configuration memory use more conservative designs than would be used in bulk SRAM. Configuration memory is slower to read or write than commodity SRAM in order to make the memory state more stable.

Although the configuration data is typically presented to the chip in serial mode in order to conserve pins, configuration is not shifted into the chip serially in modern FPGAs. If a shift register is directly connected to the programmable interconnection points and I/O pins, then the connections will constantly change as the configuration is shifted through. Many of these intermediate states will cause drivers to be shorted together, damaging the chip. Configuration bits are shifted into a temporary register and then written in parallel to a block of configuration memory [Tri98].

scan chains and JTAG

Many modern FPGAs incorporate their reconfiguration **scan chains** into their testing circuitry. Manufacturing test circuitry is used to ensure that the chip was properly manufactured and that the board on which the chip is placed is properly manufactured. The **JTAG** standard (JTAG stands for Joint Test Action Group) was created to allow chips on boards to be more easily tested. JTAG is often called **boundary scan** because it is designed to scan the pins at the boundary between the chip and the board. As shown in Figure 3-7, JTAG is built into the pins of the chip. During testing, the pins can be decoupled from their normal functions and used as a shift register. The shift register allows input values to be placed on the chip's pins and output values to be read from the pins. The process is controlled by the **test access port (TAP)** controller. The controller is connected to four pins: TDI, the shift register input; TDO, the shift register output; TCK, the test clock; and TMS, test mode select. (The standard also allows an optional test reset pin known as TRST.) The test access port includes an instruction register (IR) that determines what actions are taken by the TAP; the standard defines the state transition graph for the TAP's function. A bypass register (BP) allows bits to be either shifted into the IR or for the IR's contents to be left intact. Each pin on the chip is modified to include the JTAG shift register logic.

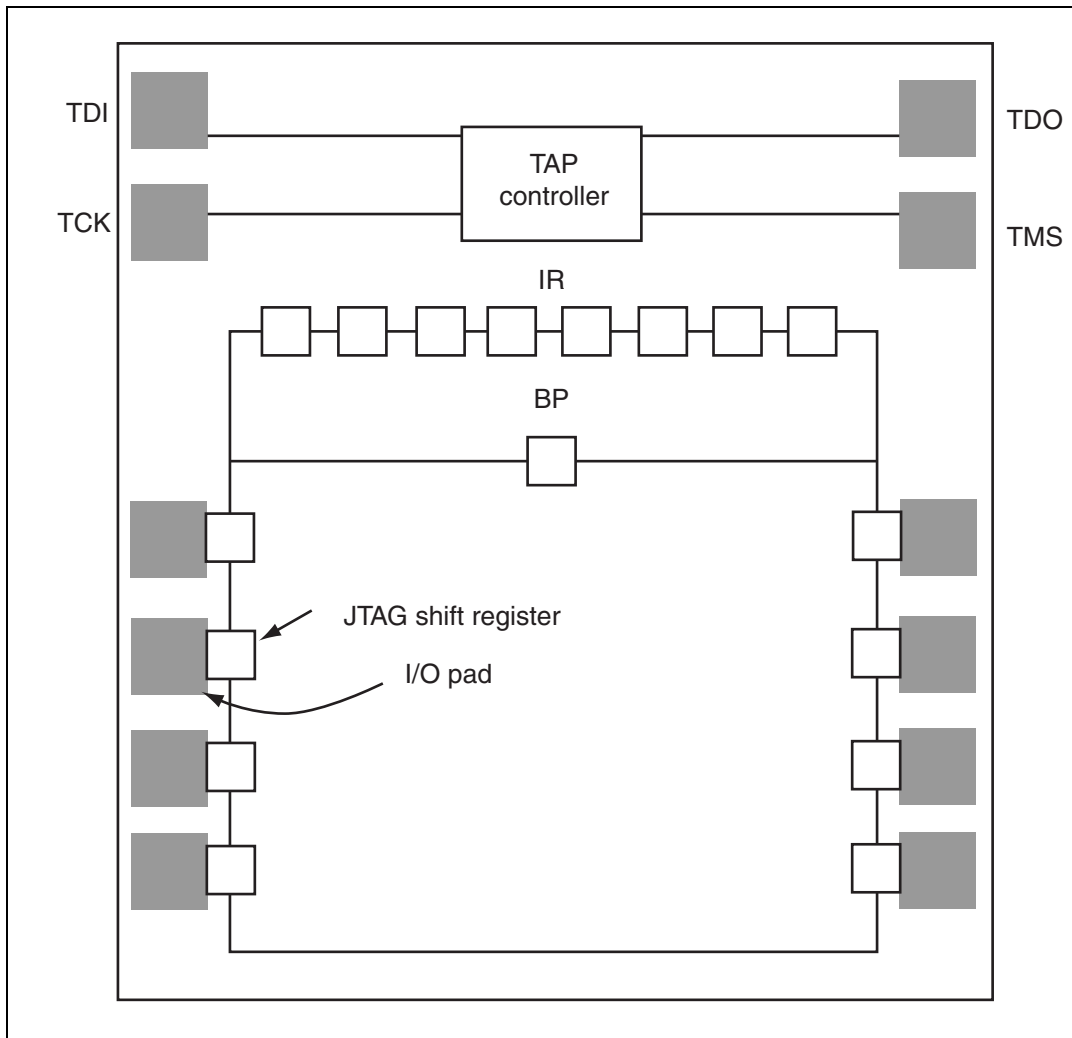


Figure 3-7 The JTAG architecture.

Using this relatively small amount of logic, an outside unit can control and observe all the pins on the chip.

The next two examples discuss the configuration systems for the Spartan-II and APEX-II.

Example 3-6
Xilinx Spartan-II
configuration

The Spartan-II configuration requires, depending on the size of the chip, from about 200,000 to over 1.3 million bits.

The chip can be configured in one of several modes:

- Master serial mode assumes that the chip is the first chip in a chain (or the only chip). The master chip loads its configuration from an EPROM or a download cable.
- Slave serial mode gets its configuration from another slave serial mode chip or from the master serial mode chip in the chain.
- Slave parallel mode allows fast 8-bit-wide configuration.
- Boundary scan mode uses the standard JTAG pins.

Several pins are dedicated to configuration. The PROGRAM[†] pin carries an active-low signal that can be used to initiate configuration. The configuration mode is controlled by three pins M0, M1, and M2. The DONE pin signals when configuration is finished. The boundary scan pins TDI, TDO, TMS, and TCK can be used to configure the FPGA without using the dedicated configuration pins.

Example 3-7
Altera APEX-II
configuration

The standard APEX-II configuration stream is one bit wide. The configuration can be supplied by a ROM or by a system controller such as a microprocessor. The APEX-II can be configured in less than 100 ms.

The APEX-II also supports a byte-wide configuration mode for fast configuration.

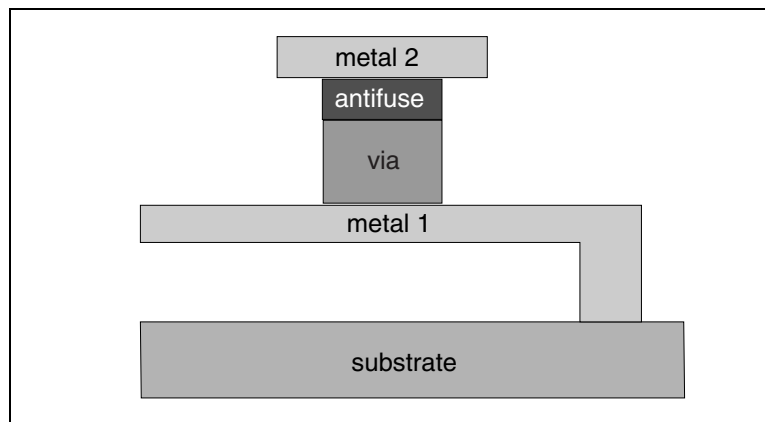
3.4 Permanently Programmed FPGAs

SRAM-based FPGAs have to be configured at power-up. There are two technologies used to build FPGAs that need to be configured only once: antifuses and flash. In this section we will survey methods for using both to build FPGAs.

3.4.1 Antifuses

An antifuse, as shown in Figure 3-8, is fabricated as a normally open disconnection. When a programming voltage is applied across the antifuse, it makes a connection between the metal line above it and the via to the metal line below. An antifuse has a resistance on the order of $100\ \Omega$, which is more resistance than a standard via. The antifuse has several advantages over a fuse, a major one being that most connections in an FPGA should be open, so the antifuse leaves most programming points in the proper state.

Figure 3-8 Cross-section of an antifuse.



An antifuse is programmed by putting a voltage across it. Each antifuse must be programmed separately. The FPGA must include circuitry that allows each antifuse to be separately addressed and the programming voltage applied.

3.4.2 Flash Configuration

Flash memory is a high-quality programmable read-only memory. Flash uses a floating gate structure in which a low-leakage capacitor holds a voltage that controls a transistor gate. This memory cell can be used to control programming transistors.

Figure 3-9 shows the schematic of a flash-programmed cell. The memory cell controls two transistors. One is the programmable connection point. It can be used for interconnect electrical nodes in interconnect or logic. The other allows read-write access to the cell.

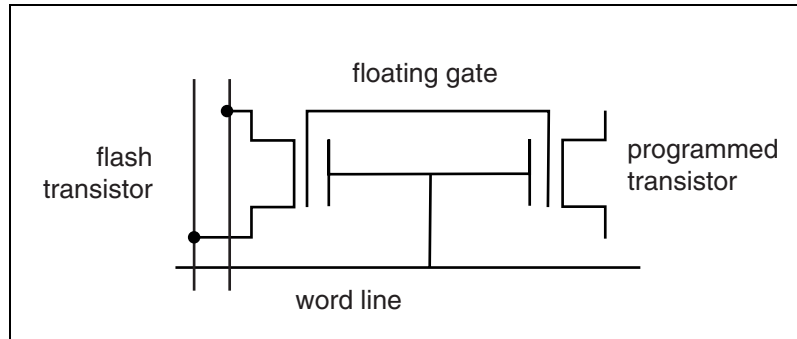


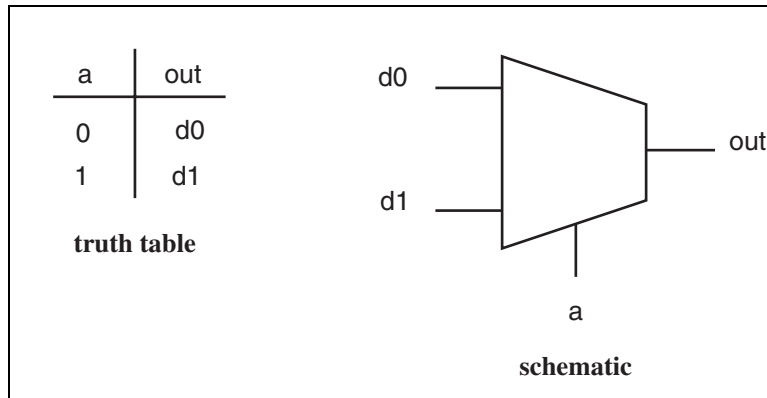
Figure 3-9 A flash programmed switch.

3.4.3 Logic Blocks

multiplexers and programming

The logic blocks in antifuse-programmed FPGAs are generally based upon multiplexing, since that function can be implemented by making or breaking connections and routing signals.

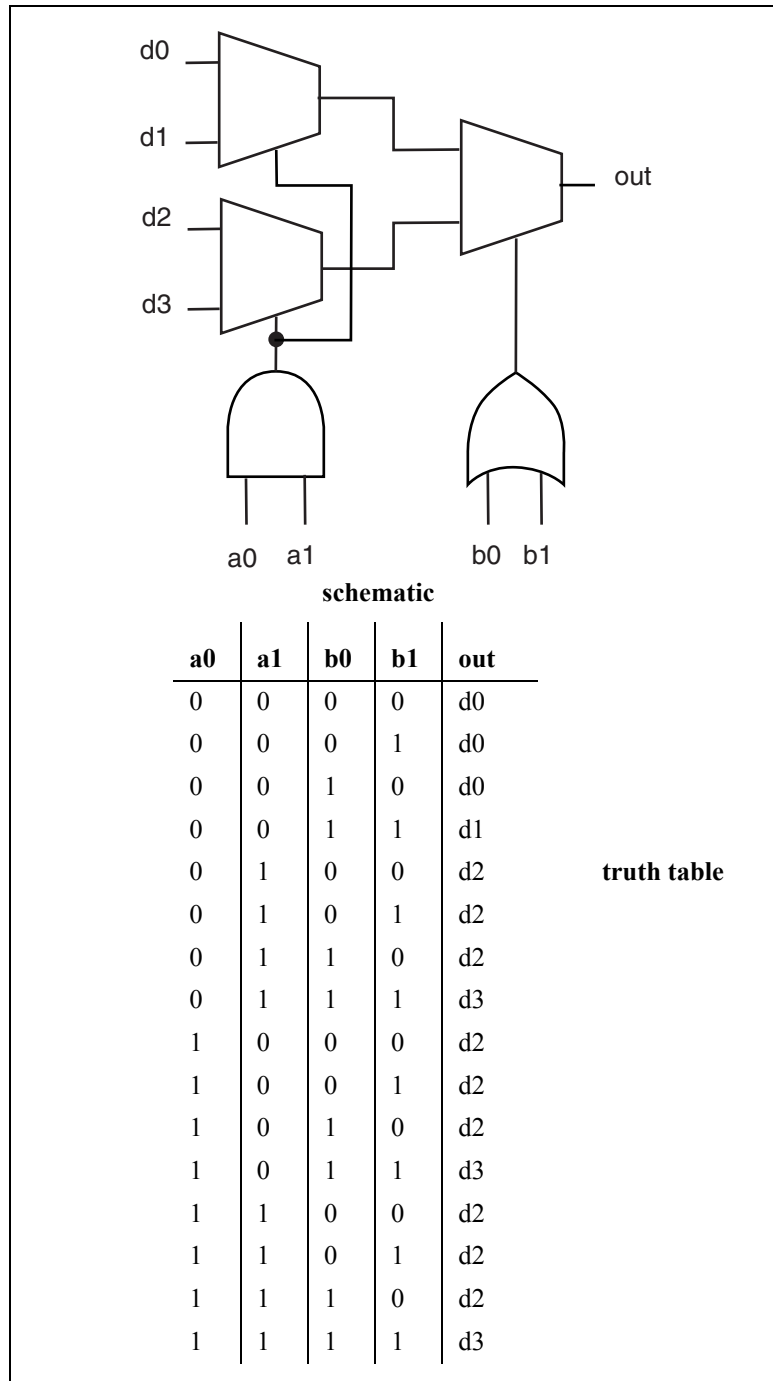
Figure 3-10 A single multiplexer used as a logic element.



To understand implementing logic functions with multiplexers, consider first the circuit of Figure 3-10. When the multiplexer control *a* is 0, the output is *d0*; when the control is 1, the output is *d1*. This logic element lets us configure which signal is copied to the logic element output. We can write the logic element's truth table as shown in the figure.

Now consider the more complex logic element of Figure 3-11. This element has two levels of multiplexing and four control signals. The con-

Figure 3-11 A logic element built from several multiplexers.



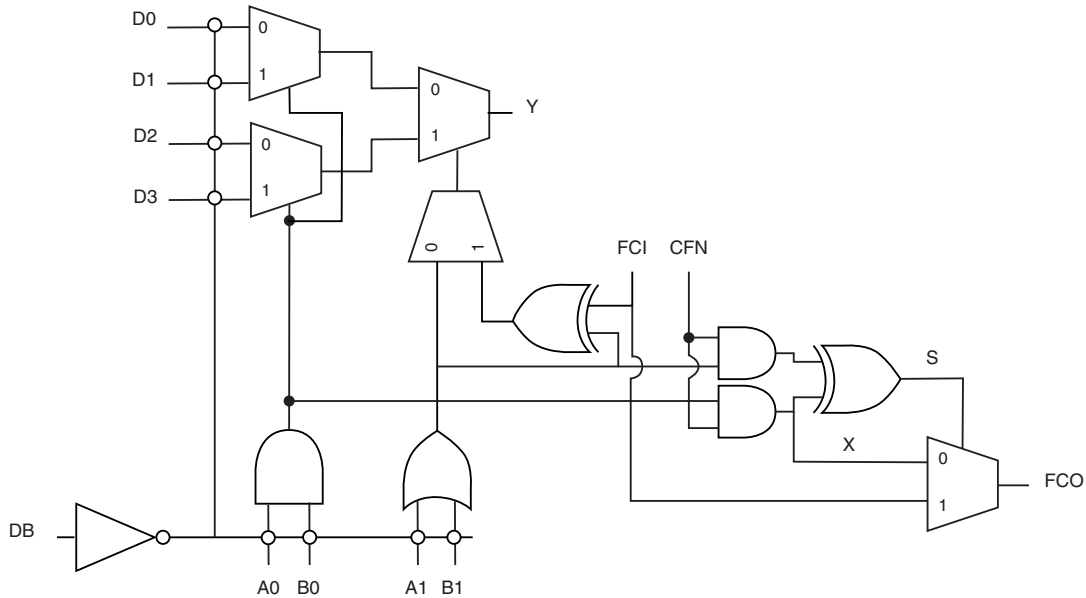
trol signals of both multiplexers in the first row are controlled by the same signal, which is the AND of two control signals. The final multiplexer stage is controlled by the OR of two other control signals. This provides a significantly more complex function.

The next example describes the logic blocks of the Actel Accelerator family FPGA [Act02]. Members of this family range in capacity from 80,000 to 1 million gates.

Example 3-8
Actel
Accelerator
family logic
elements

The Actel Accelerator family has two types of logic elements: the **C-cell** for combinational logic and the **R-cell** for registers. These cells are organized into SuperClusters, each of which has four C-cells, two R-cells, and some additional logic.

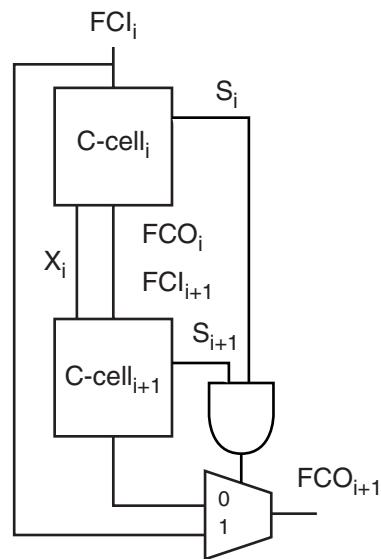
Here is the C cell:



The core of the cell is the multiplexer, which has four data signals: *D0*, *D1*, *D2*, *D3*; and four select signals: *A0*, *A1*, *A2*, and *A3*. The *DB* input can be used to drive the inverted form of any of these signals to the multiplexer by using an antifuse to connect the signal to the *DB* input, then

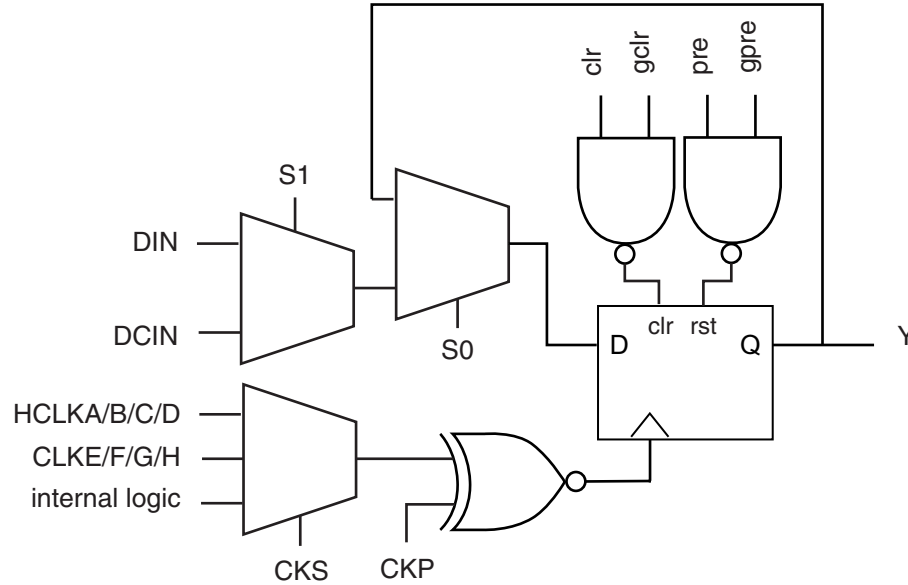
using one of the antifuses shown to connect the inverted signal to the desired multiplexer input. The signals can also be connected in their uncomplemented form. The S and X bits are used for fast addition and are not available outside the SuperCluster.

The cell includes logic for fast addition. The two bits to be added arrive at the $A0$ and $A1$ inputs. The cell has a carry input FCI and carry output FCO . The carry logic is active when the CFN signal is high. The carry logic operates in a group of two C-cells in a SuperCluster:



Within $C\text{-cell}_{i+1}$, the X_i bit is connected to the 1 input of the FCO multiplexer in this configuration. The Y output of $C\text{-cell}_{i+1}$ is used as the sum. This logic performs a carry-skip operation for faster addition.

The R-cell allows for various combinations of register inputs, clocking, etc.:



DCIN is a hardwired connection to the *DCOUT* signal of the adjacent *C-cell*; this connection is designed to have less than 0.1 ns of wire delay. The *S0* and *S1* inputs act as data enables. The flip-flop provides active low clear and presets, with clear having higher priority. A variety of clock sources can be selected by the *CKS* control signal; the *CKP* signal selects the polarity of the clock used to control the flip-flop.

Each SuperCluster has two clusters. Each cluster has three cells in the pattern CCR.

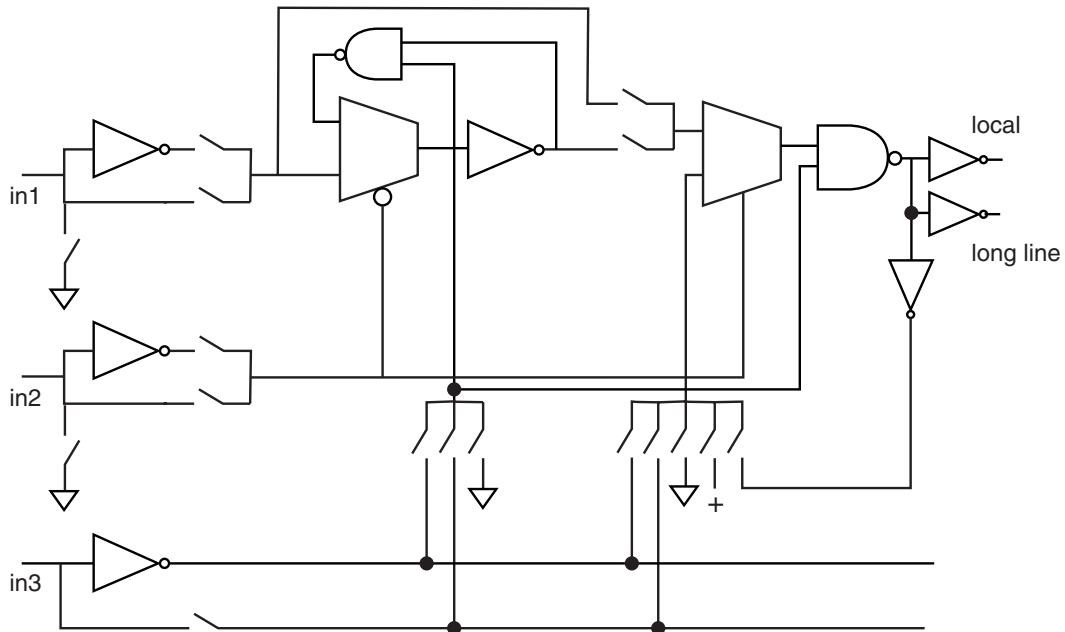
logic elements for flash-based FPGAs

Flash-based FPGAs use switches for programmability. The next example describes the Actel ProASIC 400K family logic element [Act03b].

Example 3-9
ProASIC 500K
logic element

The Actel ProASIC 500K logic element uses multiplexers to generate the desired logic function. The programmed switches are used to select alternate inputs to the core logic.

Here is the core logic tile:



Each of the three inputs can be presented to the multiplexers in true or complement form. The multiplexer system can implement any function of three inputs except for three-input XOR. The feedback paths allow the logic element to be configured as a latch, in which case *in2* is used as the clock and *in3* as reset. The logic element provides two output drivers, one for local interconnect and a larger driver for long lines.

3.4.4 Interconnection Networks

Antifuses make it relatively easy to program interconnect. An antifuse also slows down the interconnect path less than a pass transistor in an SRAM-programmable FPGA.

The next example describes the wiring organization of the Actel Accelerator family [Ac02].

Example 3-10
Actel
Axcelerator
interconnect
system

The Axcelerator has three different local wiring systems. The FastConnect system provides horizontal connections between logic modules within a SuperCluster or to the SuperCluster directly below. CarryConnects route the carry signals between SuperClusters. DirectConnect connects entirely within a SuperCluster—it connects a C-cell to the neighboring R-cell. A DirectConnect signal path does not include any antifuses; because it has lower resistance it runs faster than programmable wiring.

Generic global wiring is implemented using segmented wiring channels. Routing tracks run across the entire chip both horizontally and vertically. Although most of the wires are segmented with segments of several different lengths, a few wires run the length of the chip.

The chip provides three types of global signals. Hardwired clocks (HCLK) can directly drive the clock input of each R-cell. Four routed clocks can drive the clock, clear, preset, or enable pin of an R-cell or any input of a C-cell. Global clear (GCLR) and global preset (GPSET) signals drive the clear and preset signals of R-cells and I/O pins.

interconnect in flash-based FPGAs

The next example describes the interconnect structure of the ProASIC 500K [Act02].

Example 3-11
Actel ProASIC
500K
interconnect
system

The ProASIC 500K provides local wires that allow the output of each tile to be directly connected to the eight adjacent tiles. Its general-purpose routing network uses segmented wiring with segments of length 1, 2, and 4 tiles. The chip also provides very long lines that run the length of the chip.

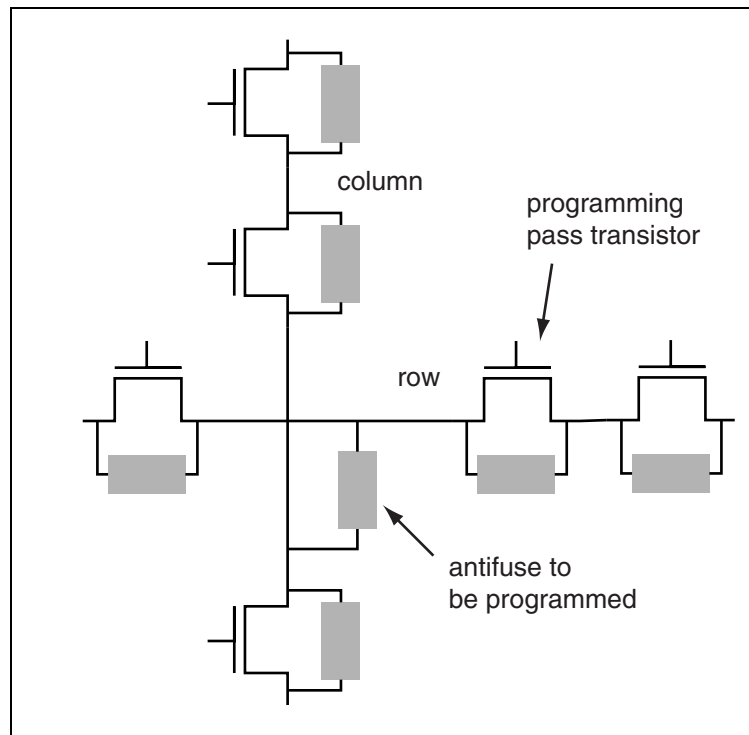
The ProASIC 500K provides four global networks. Each network can be accessed either from a dedicated global I/O pin or a logic tile.

3.4.5 Programming

antifuse programming

An antifuse is programmed by applying a large voltage sufficient to make the antifuse connection. The voltage is applied through the wires connected by the antifuse. The FPGA is architected so that all the antifuses are in the interconnect channels; this allows the wiring system to be used to address the antifuses for programming. In order to be sure

Figure 3-12 Selecting an antifuse to be programmed.



that every antifuse can be reached, each antifuse is connected in parallel with a pass transistor that allows the antifuse to be bypassed during programming. The gates of the pass transistors are controlled by programming signals that select the appropriate row and column for the desired antifuse, as shown in Figure 3-12. The programming voltage is applied across the row and column such that only the desired antifuse receives the voltage and is programmed [EIG98].

Because the antifuses are permanently programmed, an antifuse-based FPGA does not need to be configured when it is powered up. No pins need to be dedicated to configuration and no time is required to load the configuration.

3.5 Chip I/O

features of I/O pins

The I/O pins on a chip connect it to the outside world. The I/O pins on any chip perform some basic functions:

- Input pins provide electrostatic discharge (ESD) protection.
- Output pins provide buffers with sufficient drive to produce adequate signals on the pins.
- Three-state pins include logic to switch between input and output modes.

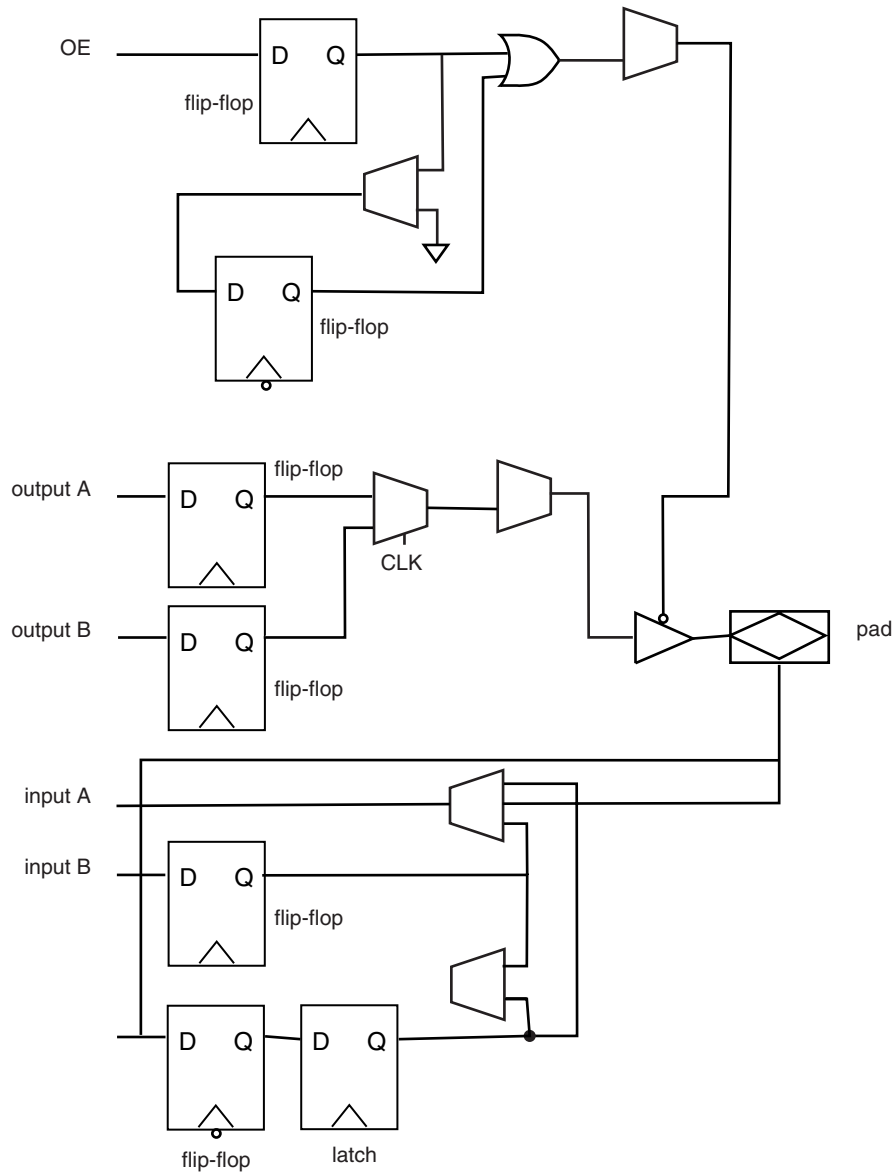
The pins on an FPGA must be programmable to accommodate the requirements of the configured logic. A standard FPGA pin can be configured as either an input, output, or three-state pin.

Pins may also provide other features. Registers are typically provided at the pads so that input or output values may be held. The slew rate of outputs may be programmable to reduce electromagnetic interference; lower slew rates on output signals generate less energetic high-frequency harmonics that show up as electromagnetic interference (EMI).

The next two examples describe the I/O pins of the Actel APEX-II and the Xilinx Spartan-II 2.5V FPGA.

Example 3-12 Altera APEX-II I/O pin

The Altera APEX-II I/O structure, known as an IOE, is designed to support SDRAM and double-data-rate (DDR) memory interfaces. It contains six registers and a latch as well as bidirectional buffers. The IOE supports two inputs and two outputs:



The OE signal controls the three-state behavior.

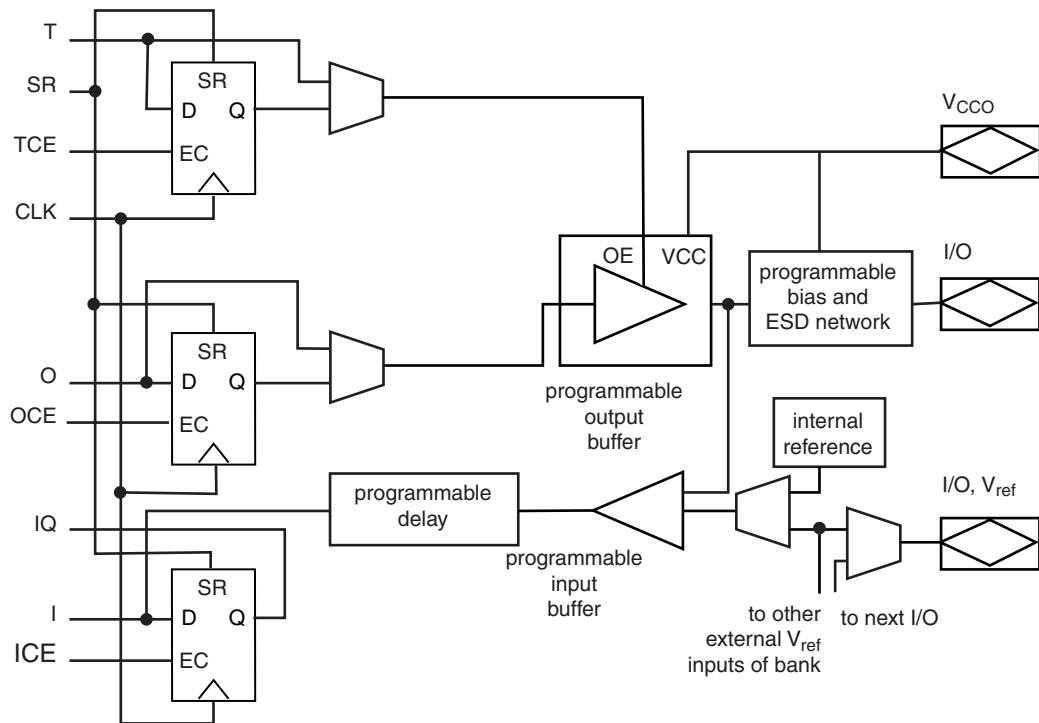
Example 3-13

The Xilinx Spartan-II 2.5V I/O pin

The Spartan-II 2.5V family is designed to support a wide variety of I/O standards:

I/O standard	Input reference voltage (V_{ref})	Output source voltage (V_{CCO})	Board termination voltage (V_{TT})
LVTTL	N/A	3.3	N/A
LVC MOS2	N/A	2.5	N/A
PCI	N/A	3.3	N/A
GTL	0.8	N/A	1.2
GTL+	1.0	N/A	1.5
HSTL Class I	0.75	1.5	0.75
HSTL Class III	0.9	1.5	1.5
HSTL Class IV	0.9	1.5	1.5
SSTL3 Class I and II	1.5	3.3	1.5
SSTL2 Class I and II	1.25	2.5	1.25
CTT	1.5	3.3	1.5
AGP-2X	1.32	3.3	N/A

Here is the schematic for the I/O block:



Much of the right-hand side of the schematic is devoted to handling the various I/O standards. Notice that pins are required for the various reference voltages as well as the I/O itself. The pins on the chip are divided into eight banks, with each bank sharing the reference voltage pins. Pins within a bank must use standards that have the same V_{CC0} .

The IOB has three registers, one each for input, output, and three-state operation. Each has its own enable (ICE, OCE, and TCE, respectively) but all three share the same clock connection. These registers in the IOB can function either as flip-flops or latches.

The programmable delay element on the input path is used to eliminate variations in hold times from pin to pin. Propagation delays within the FPGA cause the IOB control signals to arrive at different times, causing the hold time for the pins to vary. The programmable delay element is matched to the internal clock propagation delay and, when enabled, eliminates skew-induced hold time variations.

The output path has a weak keeper circuit that can be selected by programming. The circuit monitors the output value and weakly drives it to the desired high or low value. The weak keeper is useful for pins that are connected to multiple drivers; it keeps the signal at its last valid state after all the drivers have disconnected.

3.6 Circuit Design of FPGA Fabrics

Circuit design determines many of the characteristics of FPGA architectures. The size of a logic element determines how many can be put on a chip; the delay through a wire helps to determine the interconnection architecture of the fabric. In this section we will look at the circuit design of both logic elements and interconnections in FPGA fabrics, primarily concentrating on SRAM-based FPGAs, but with some notes on antifuse-based FPGAs. We will rely heavily on the results of Chapter 2 throughout this section.

3.6.1 Logic Elements

LEs vs. logic gates

The logic element of an FPGA is considerably more complex than a standard CMOS gate. A CMOS gate needs to implement only one chosen logic function. The logic element of an FPGA, in contrast, must be able to implement a number of different functions.

Antifuse-based FPGAs program their logic elements by connecting various signals, either constants or variables, to the inputs of the logic elements. The logic element itself is not configured as a SRAM-based logic element would be. As a result, the logic element for an antifuse-based FPGA can be fairly small. Figure 3-13 shows the schematic for a multiplexer-based logic element used in early antifuse-based FPGAs. Table 3-1 shows how to program some functions into the logic element by connecting its inputs to constants or signal variables. The logic element can also be programmed as a dynamic latch.

Example 3-14 compares lookup tables and static gates in some detail.

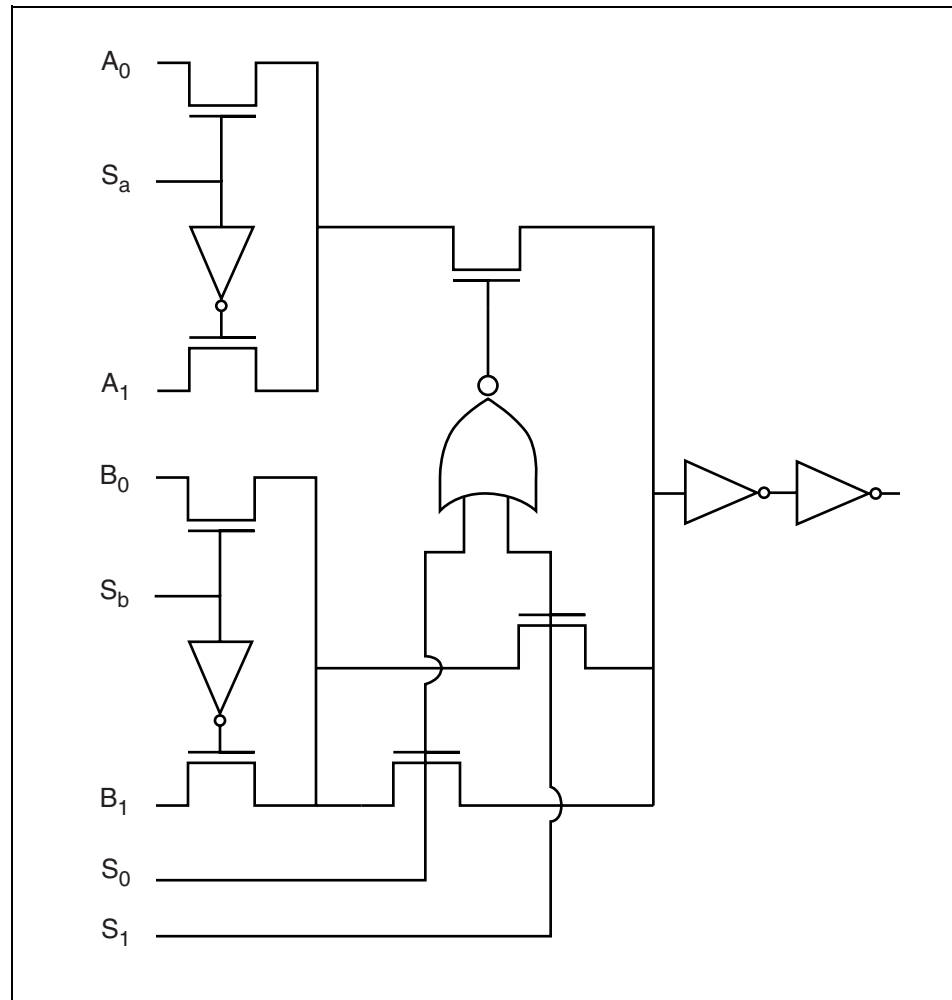


Figure 3-13 A logic element built from a multiplexer [EIG90].

equation	A₀	A₁	B₀	B₁	S_a	S₁	S₀	S_b
(AB)'	1	1	0	1	A	0	B	A
	0	1	0	1	0	0	B	A
	0	1	0	1	0	B	0	A
	0	1	0	1	0	0	A	B
A^B	1	0	0	1	A	0	B	A
	1	0	0	1	A	B	0	A
latch	Q	0	D	0	CLR	CLK	0	CLR
	Q	0	CLR	0	CLR	CLK	0	D

Table 3-1 Programming the mux-based logic element [EIG90].

Example 3-14
Lookup table vs.
static CMOS gate

Let us compare the static CMOS gate and lookup table logic element in several respects: size, delay, and power consumption.

For our purposes, counting the number of transistors in a circuit gives us a sufficiently accurate estimate of the size of the circuit on-chip. The number of transistors in a static CMOS gate depend on both the number of inputs to the gate and the function to be implemented. A static NAND or NOR gate of n inputs has $2n$ transistors; more complex functions such as XOR can be more complicated. A single NAND or NOR gate with 16 inputs is impractical, but it would require 32 transistors.

In contrast, the SRAM cell in the lookup table requires eight transistors, including the configuration logic. For a four-input function, we would have $8 \times 16 = 128$ transistors just in the core cell. In addition, we need decoding circuitry for each bit in the lookup table. A straightforward decoder for the four-bit lookup table would be a multiplexer with 96 transistors, though smaller designs are possible.

The delay of a static gate depends not only on the number of inputs and the function to be implemented, but also on the sizes of transistors used. By changing the sizes of transistors, we can change the delay through the gate. The slowest gate uses the smallest transistors. Using logical effort theory [Sut99], we can estimate the delay of a chain of two four-input NAND gates that drives another gate of the same size as 9τ units.

The delay of a lookup table is independent of the function implemented and dominated by the delay through the SRAM addressing logic. Logical effort gives us the decoding time as 21τ units.

The power consumption of a CMOS static gate is, ignoring leakage, dependent on the capacitance connected to its output. The CMOS gate consumes no energy while the inputs are stable (once again, ignoring leakage). The SRAM, in contrast, consumes power even when its inputs do not change. The stored charge in the SRAM cell dissipates slowly (in a mechanism independent of transistor leakage); that charge must be replaced by the cross-coupled inverters in the SRAM cell.

As we can see, the lookup table logic element is considerably more expensive than a static CMOS gate.

Because the logic element is so complex, its design requires careful attention to circuit characteristics. In this section we will concentrate on lookup tables—most of the complexity of an antifuse-based logic element is in the antifuse itself, and the surrounding circuitry is highly

dependent on the electrical characteristics of the antifuse material. The lookup table for an SRAM-based logic element incorporates both the memory and the configuration circuit for that memory.

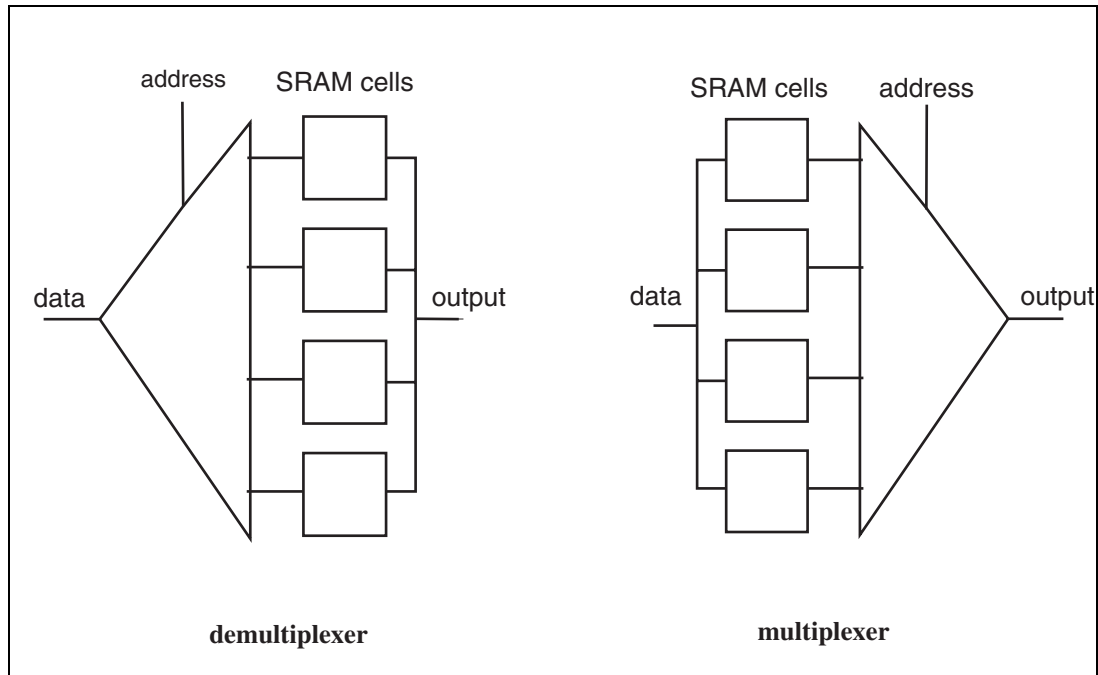


Figure 3-14 Organizations for lookup table addressing.

SRAMs for LEs

There are two possible organizations for the lookup table as shown in Figure 3-14: a demultiplexer that causes one bit to drive the output or a multiplexer that selects the proper bit. These organizations are logically equivalent but have different implications for circuitry. Bulk SRAMs generally use the demultiplexer architecture, as shown in Section 2.6.2. The demultiplexer selects a row to be addressed, and the shared bit lines are used to read or write the memory cells in that row. The shared bit line is very efficient in large memories but less so in small memories like those used in logic elements. Most FPGA logic elements use a multiplexer to select the desired bit.

Most large SRAMs use two bit lines that provide complementary values of the SRAM cell value. Using bit and bit-bar lines improves access times in large SRAMs but does not make any noticeable improvement in small SRAMs used in logic elements. As a result, logic element SRAM

cells are generally read and written through only one side of the cell, not through both sides simultaneously.

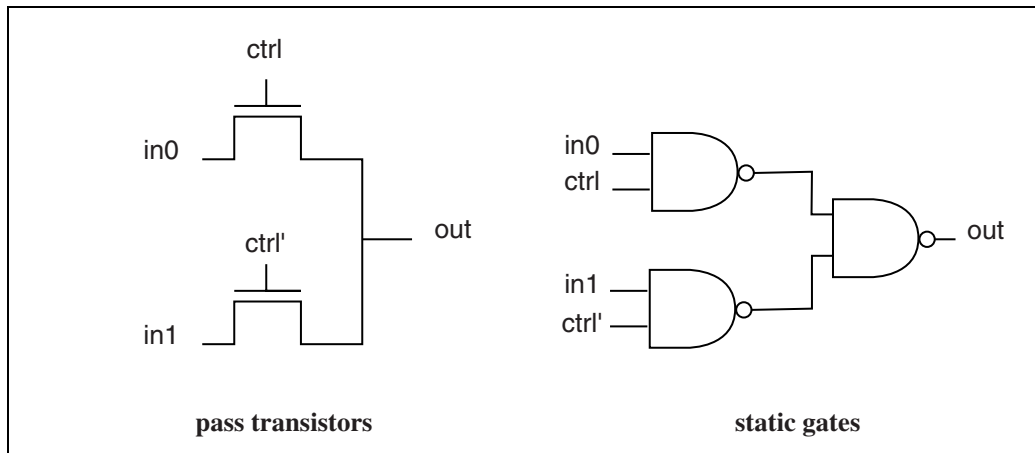


Figure 3-15 Alternative circuits for a multiplexer.

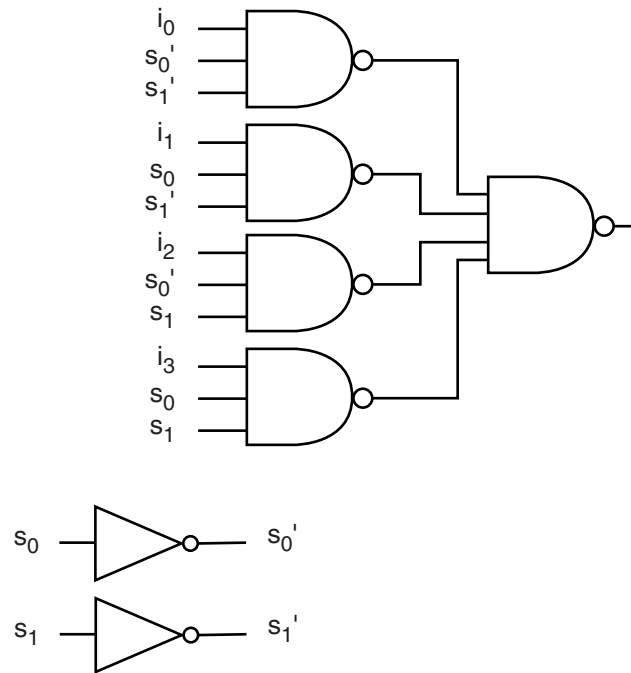
SRAM multiplexer design

Should that multiplexer be made of static gates or pass transistors? The alternatives for the case of a two-input multiplexer are shown in Figure 3-15. The pass transistor network is clearly smaller—each two-input NAND or NOR gate has four transistors. But as the number of series pass transistors grows the delay from the data input to the data output grows considerably. The delay through a series of pass transistors, in fact, grows as the square of the number of pass transistors in the chain, for reasons similar to that given by Elmore. The choice between static gates and pass transistors therefore depends on the size of the lookup table. The next example compares the delay through static gate and pass transistor multiplexers.

Example 3-15 **Delay through** **multiplexer** **circuits**

We want to build a b -input multiplexer that selects one of the b possible input bits. We will call the data input bits i_0 , *etc.* and the select bits s_0 , *etc.* In our drawings we will show four-input multiplexers; these are smaller than the multiplexers we want to use for lookup tables but they are large enough to show the form of the multiplexer.

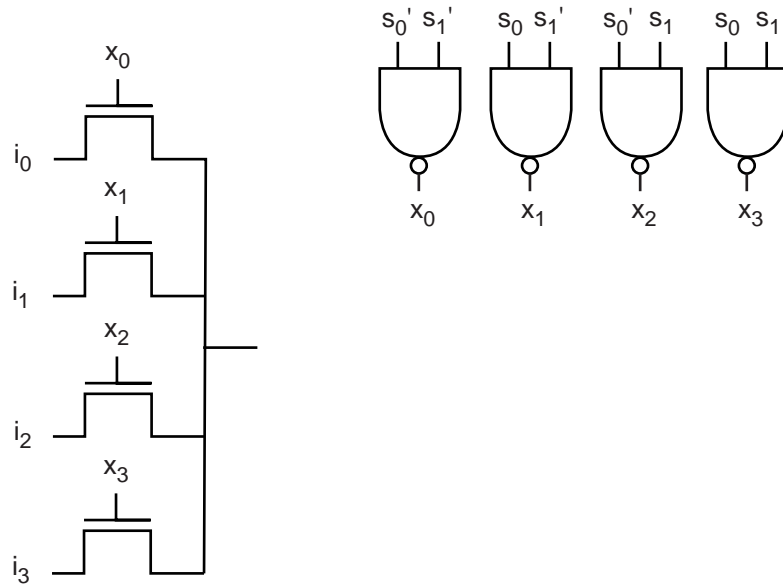
Here is a four-input mux built from NAND gates:



This multiplexer uses two levels of logic plus some inverters that form a third level of logic. Each of the NAND gates in the first level of logic have as inputs one of the data bits and true or complement forms of all the select bits. The inverters are used to generate the complement forms of the select bits. Each NAND gate in the first level determines whether to propagate the input data bit for which it is responsible; the second-level NAND sums the partial results to create a final output.

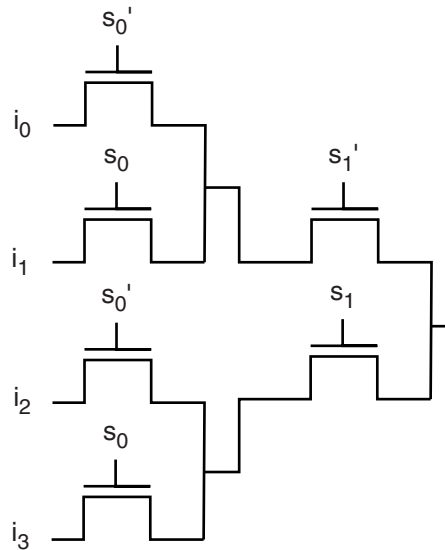
We can analyze the delay through a b -bit multiplexer as a function of b using logical effort [Sut99]. The delay through an n -input NAND gate is proportional to $(n + 2)/3$. Each NAND gate in the first level of logic has $\lg b$ inputs for the select bits and one for the data input, giving a delay proportional to $(\lg b + 3)/3$. The second-level NAND gate has b inputs, for delay proportional to $(b + 2)/3$. The delay through the inverters on the select bits is proportional to 1. This means that the total delay through the b -bit static gate multiplexer grows as $b \lg b$.

Here is one form of multiplexer built from pass transistors:



While this form may seem simple and fast, the pass transistors are not the only transistors in the multiplexer. The gates must be driven by decoded address signals generated from the select bits. This circuit is not good for large multiplexers because it combines the worst aspects of static gate and pass transistor circuits.

A better form of circuit built from pass transistors is a tree:



The gates of these pass transistors are driven directly by select bits or the complements of the select bits (generated by inverters), eliminating the decoder NAND gates. However, because the pass transistors can be (roughly) modeled as resistors, the delay through a chain of pass transistors is proportional to the square of the number of switches on the path. (We analyzed delay through RC chains in Section 2.5.4.) The tree for a b -input multiplexer has $\lg b$ levels of logic, so the delay through the tree is proportional to $\lg b^2$.

One question that can be asked is whether transmission gates built from parallel n-type and p-type transistors are superior to pass transistors (that is, single n-type transistors). While transmission gates are more egalitarian in the way they propagate logic 0 and 1 signals, their layouts are also significantly larger. Chow et al. [Cho99] found that pass transistors were the better choice for multiplexers.

It is possible to build a mux from a combination of pass transistors and static gates, using switches for some of the select stages and static gates for the remaining stages.

LE output drivers

The output of the logic element must be designed with drivers sufficient to drive the interconnect at the LE's output. The transistors must be

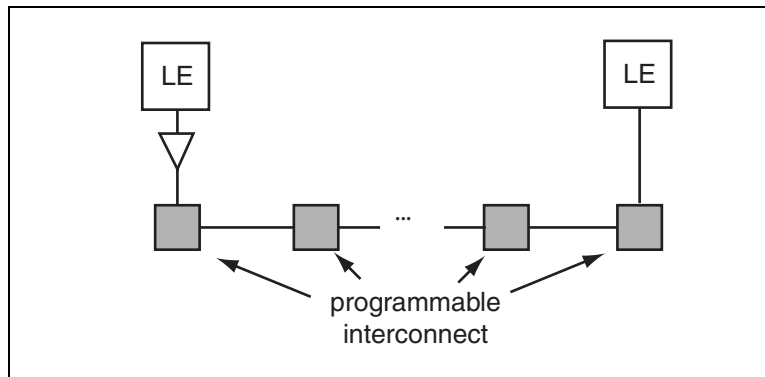
sized properly based upon a given load on the output and a desired rise/fall time. Once the characteristics of the interconnect are known, sizing the output transistors is straightforward. However, because the LE may need to drive a long wire that also includes logic used to make programmable connections, the output buffer must be powerful and large.

3.6.2 Interconnect

varieties of interconnect

The first question we need to ask about FPGA interconnect is why an FPGA has so many different kinds of interconnect. A typical FPGAs has short wires, general-purpose wires, global interconnect, and specialized clock distribution networks. The reason that FPGAs need different types of wires is that wires can introduce a lot of delay, and wiring networks of different length and connectivity need different circuit designs. We saw some uses for different types of interconnect when we studied existing FPGAs, but the rationale for building several types of interconnect becomes much clearer when we study the circuit design of programmable interconnect.

Figure 3-16 A generic signal path between two logic elements.



In Example 2-4 we compared the delay through logic gates and the delay through wires. We saw that a relatively short wire—a wire that is much shorter than the size of the chip—has a delay equal to the delay through a logic gate. Since many connections on FPGAs are long, thanks to the relatively large size of a logic element, we must take care to design circuits that minimize wire delay. A long wire that goes from one point to another needs a chain of buffers to minimize the delay through the wire. We studied optimal buffer sizing and buffer insertion

in Section 2.5.5. Now we must apply that general knowledge to the interconnect circuits in FPGAs.

Figure 3-17 Organization of a pass-transistor-based interconnection point.

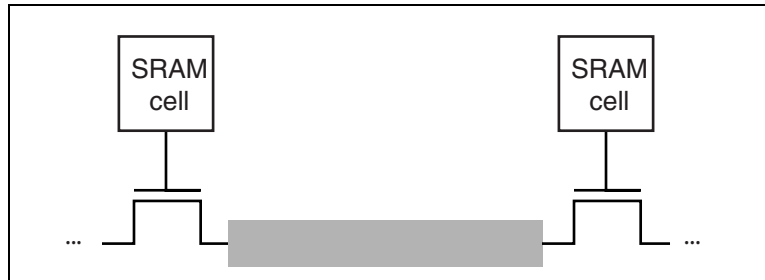


Figure 3-18 Organization of a three-state buffer-based interconnection point.

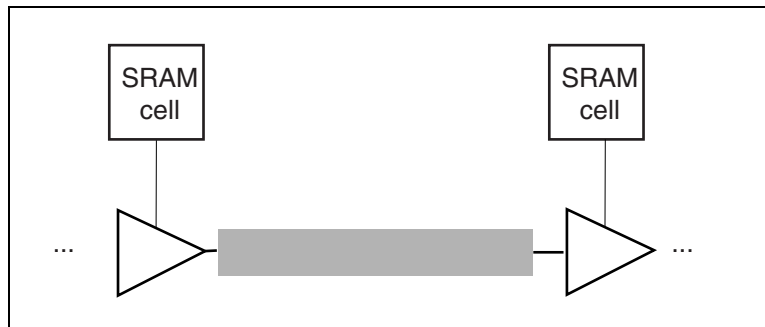


Figure 3-16 shows the general form of a path between two logic elements. A signal leaves a logic element, goes through a buffer, enters the routing channel through a programmable interconnect block, passes through several more programmable interconnect blocks, then passes through a final programmable interconnect block to enter the destination LE. We have studied the circuit design of logic elements; now we need to consider the circuits for programmable interconnect blocks. Brown et al. [Bro92] concluded that most of the area in an SRAM-based FPGA is consumed by the routing switches, so we must carefully design the programmable interconnect to make best use of the available area.

*programmable
interconnection circuit*

Antifuses provide low-impedance connections between wires. However, the design of a programmable interconnection point for an SRAM-based or flash-based FPGA requires more care because the circuitry can introduce significant delay as well as cost a significant amount of area. The circuit design of a pass-transistor-based programmable interconnection point is shown in Figure 3-17. If we use pass transistors at the programmable interconnection points, we have two parameters we can use to minimize the delay through the wire segment: the width of the pass transistor and the width of the wire. As we saw in Section 2.3, the current

through a transistor increases proportionately with its width. The increased current through the transistor reduces its effective resistance, but at the cost of a larger transistor. Similarly, we can increase the width of a wire to reduce its resistance, but at the cost of both increased capacitance and a larger wire. Rather than uniformly change the width of the wire, we can also taper the wire as described in Section 2.5.3.

We can also ask ourselves whether we should use three-state buffers rather than pass transistors at the programmable interconnection points. The use of three-state buffers in programmable interconnect is illustrated in Figure 3-18. The three-state buffer is larger than a pass transistor but it provides amplification that the pass transistor does not.

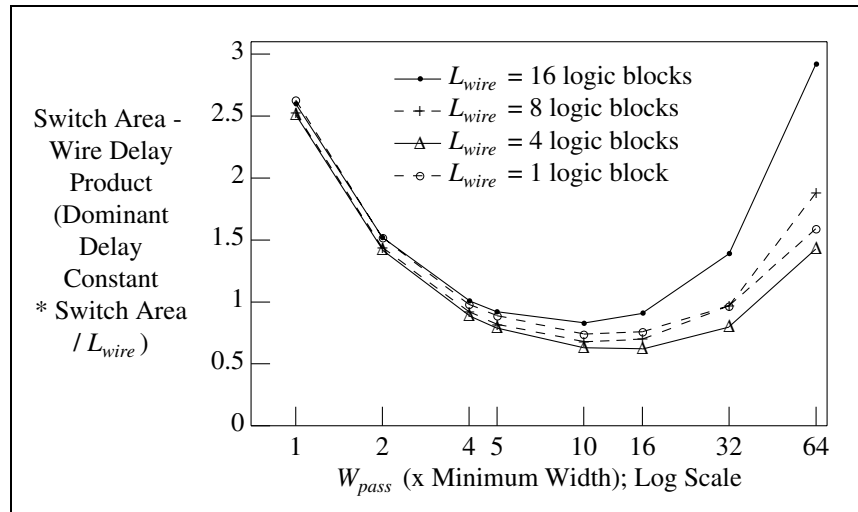


Figure 3-19 Switch area * wire delay vs. routing pass transistor width (from Betz and Rose [Betz99], © 1999 IEEE).

pass transistor and wire sizing

Betz and Rose [Betz99] considered the effects of pass transistor and wire sizing as well as the merits of three-state buffers as programmable interconnection points; their studies used a 0.35 μm technology. They use the product of area and wire delay as a metric for the cost-effectiveness of a given circuit design. Figure 3-19 compares the product of switch area and wire delay as a function of the width of the pass transistor at a programmable interconnection point. The plot shows curves for wires of different lengths, with the length of a wire being measured in multiples of the size of a logic element. The plot shows a clear minimum in the

area-delay product when the pass transistor is about ten times the minimum transistor width.

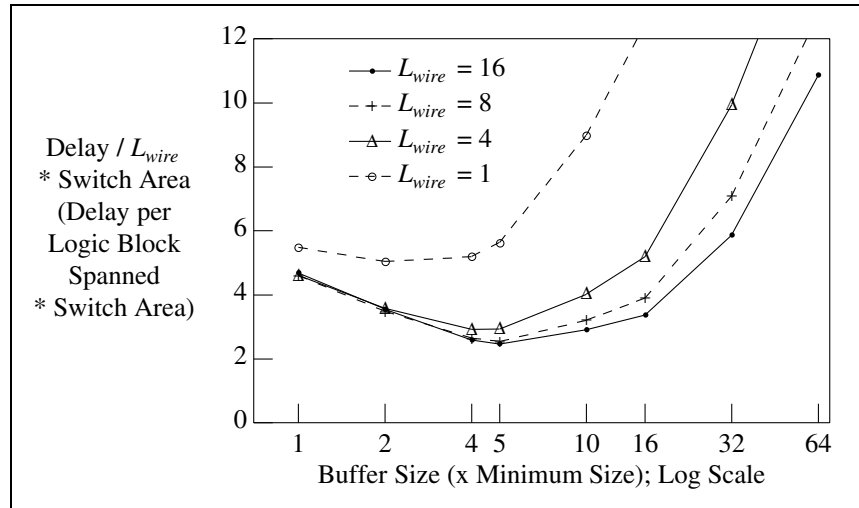


Figure 3-20 Switch area * wire delay vs. routing three-state buffer size (from Betz and Rose [Bet99], © 1999 IEEE).

Figure 3-20 shows the area-delay curve for a three-state buffer. The minimum area-delay product occurs when the three-state driver's transistors are about five times the minimum transistor size; the three-state requires smaller transistors than does the pass transistor because it provides amplification. Betz and Rose also found that increasing the width of the wire uniformly gave very little improvement in delay: doubling the wire width reduced delay by only 14%. Uniformly increasing the wire width has little effect because the wire capacitance is much larger, swamping the effects of reduced resistance.

simultaneous driver and pass transistor sizing

Chandra and Schmit [Cha02] studied the effect of simultaneously optimizing the sizes of the drivers at the LE outputs and the pass transistors in the interconnect. Figure 3-21 shows how delay through the wire varies as the driving buffer and routing switch size change; these curves were generated for a 0.18 μm technology. Each curve shows the delay for a given size of driver. The curves are U shaped—as the routing switch increases in size, delay first decreases and then increases. The initial drop in delay is due to decreasing resistance in the switch; the ultimate increase in delay happens when the increases in capacitance overwhelm the improvements obtained from lower resistance. This plot

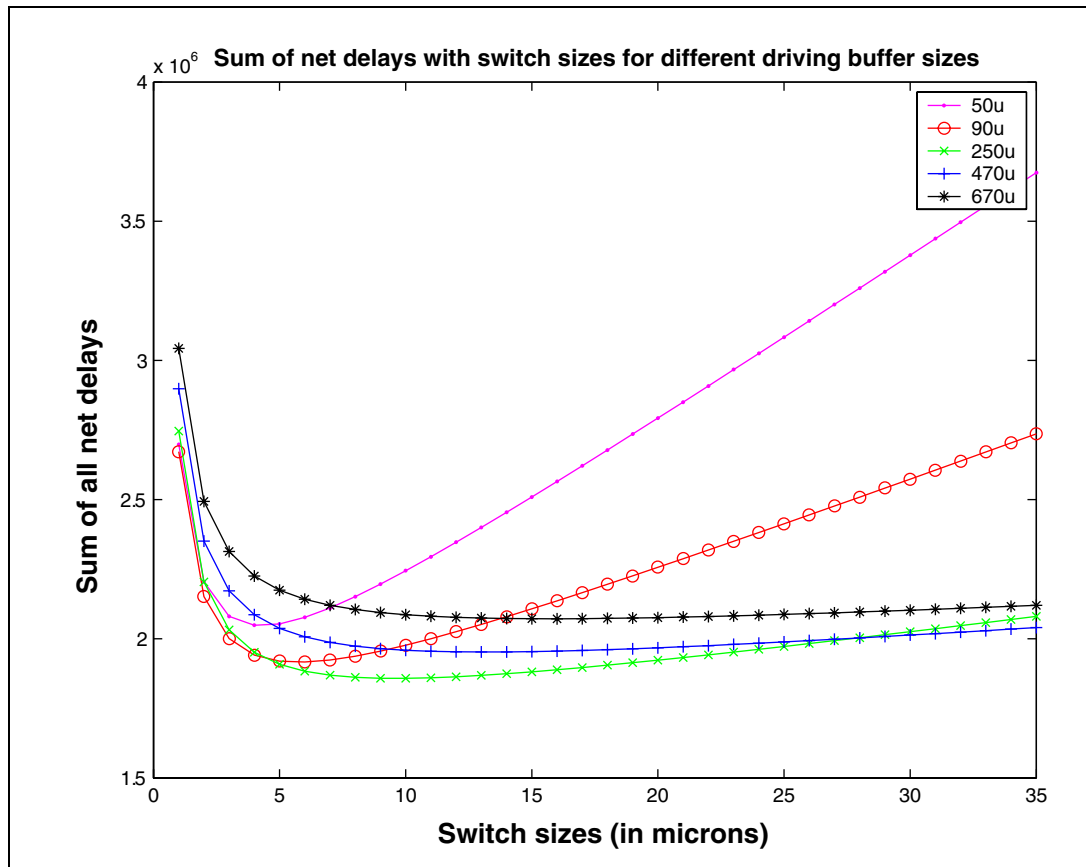
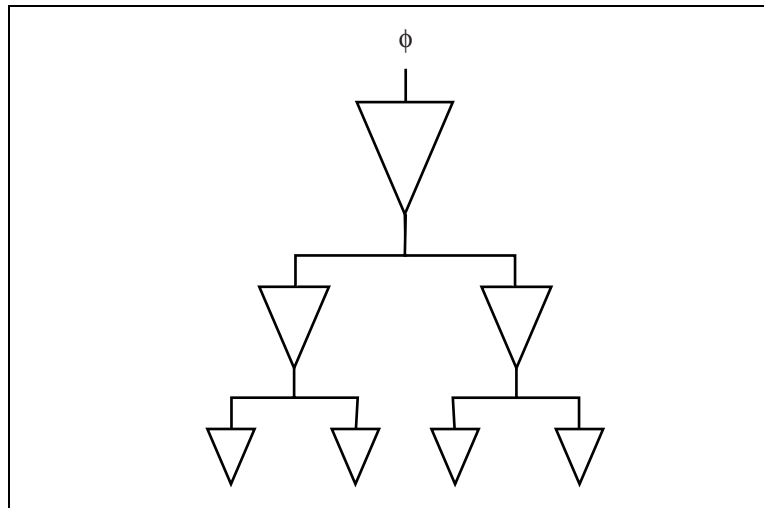


Figure 3-21 Variation of delay with routing switch and driving buffer sizes (from Chandra and Schmit [Cha02], ©2002 IEEE).

shows that there is a best size for the pass transistor routing switch for any given choice of driver size.

Figure 3-22 A clock driver tree.



clock networks

FPGAs have specialized clock wiring because the clock signal combines a large number of destinations (all the registers in use in the system) with low delays and skew between the arrival of the clock at different points on the chip. A clock network is particularly difficult because it must go to many different places. As illustrated in Figure 3-22, clock signals are often distributed by trees of drivers, with larger transistors on the drivers near the clock source and smaller transistors on the drivers close to the flip-flops and latches. This structure presents a much larger capacitive load than does a point-to-point wire. Buffers must be distributed throughout the clock tree in order to minimize delay.

3.7 Architecture of FPGA Fabrics

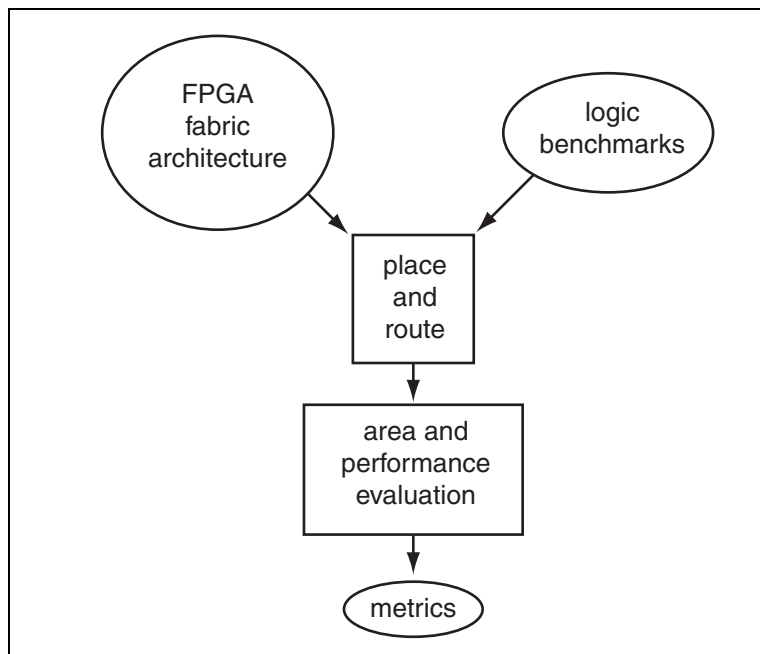
issues in fabric architecture

In addition to designing the circuits of the FPGA, we need to design the FPGA's architecture—the characteristics of the logic elements and interconnection that form the FPGA fabric.

We need to answer quite a few questions to move from the concept of an FPGA to a specific FPGA fabric:

- How many logic elements on the FPGA? Logic elements and interconnect are, to some extent, mutually exclusive, since we have only a limited amount of area on chip. Wires do exist on several levels but the transistors for the interconnection points and amplifiers take up area that could be devoted to logic elements.
- What functions should the logic element perform? How many inputs should it have? Should it provide dedicated logic for addition or other functions?
- What different types of interconnect do we need? Do we need global interconnect, local interconnect, and other types? How much of each type do we need?
- How long should interconnect segments be? Longer segments provide shorter delay but less routing flexibility.
- How should we distribute the interconnect elements? Interconnect may be distributed uniformly or in various patterns.

Figure 3-23 Methodology for evaluating FPGA fabrics.



We can answer all these questions and more using the same basic methodology shown in Figure 3-23: choose an FPGA fabric to evaluate; select a set of benchmark designs; implement the benchmarks on the test

fabric; evaluate the resulting metrics. An FPGA fabric is different from a custom chip in that it is intended to be used for many different logic designs. As a result, the standard by which the fabric should be judged is the quality of implementation of a typical set of logic designs. Companies that design FPGAs usually do not use them to build systems, so they may collect benchmarks from customers or from public sources. By implementing a set of designs and then measuring how well they fit onto the fabric, FPGA designers can get a better sense of what parts of their fabric work well and what need improvement.

We can measure the quality of a result in several ways:

- logic utilization;
- size of the logic element;
- interconnect utilization;
- area consumed by the connection boxes for the interconnect;
- worst-case delay.

This methodology works because we use computer-aided design (CAD) tools to implement logic designs on FPGAs. We will discuss CAD tools in more detail in later chapters; at the moment it is only necessary to know that we can map logic designs onto FPGAs automatically.

3.7.1 Logic Element Parameters

factors in LE design

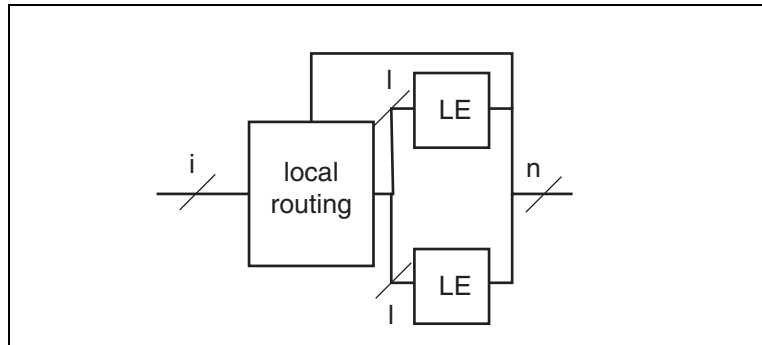
The most basic question we can ask about a lookup table logic element is the number of inputs it should have. This choice is subject to delicate trade-offs:

- If the LE has too few inputs, each lookup table and the associated interconnect have a higher proportion of overhead circuitry. Fewer transistors are devoted to logic.
- If the LE has too many inputs, the logic designs that are mapped into the LEs may not be large enough to make use of all the capacity of the lookup tables, thus wasting the capacity of the LEs.

The choice of a size for the lookup table therefore depends on both the circuit design of lookup tables and the characteristics of the designs to be implemented in the FPGAs.

Many experiments [Bro92] have found that a lookup table with four inputs (and therefore 16 entries) is the best choice.

Figure 3-24 A logic element cluster.



LE clusters

Betz and Rose [Bet98] studied logic element clusters. As shown in Figure 3-24, a logic cluster has several logic elements and some dedicated interconnect. The i inputs to the cluster are routed to the LEs and to the cluster outputs by a local routing network. Each LE has l inputs. Because this local network does not provide for full connectivity between the cluster inputs and LEs, it requires much less chip area than programmable interconnect, but it provides only certain connections. This leads one to ask whether clusters are better than monolithic logic elements and what sort of cluster configuration is best.

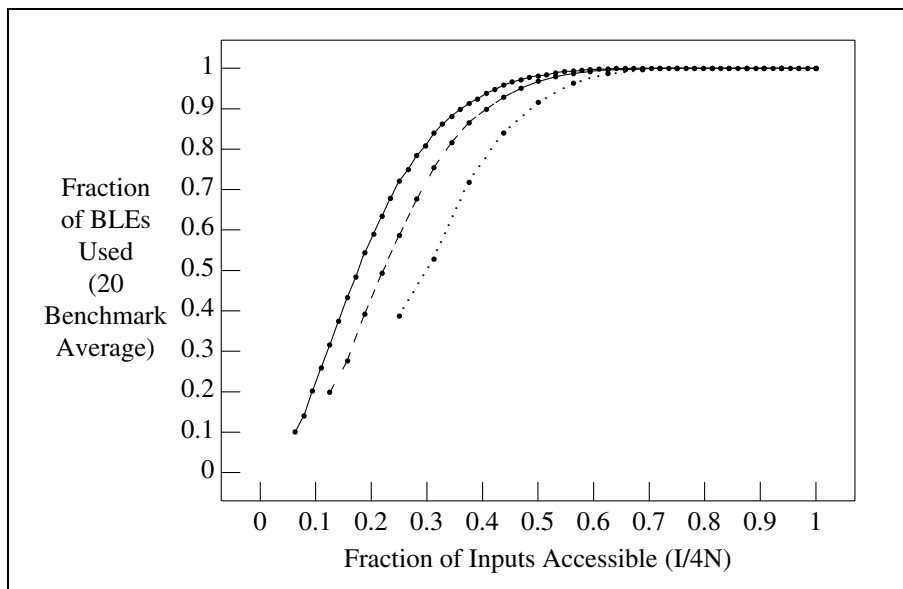


Figure 3-25 Logic utilization vs. number of logic cluster inputs (from Betz and Rose [Bet98], ©1998 IEEE).

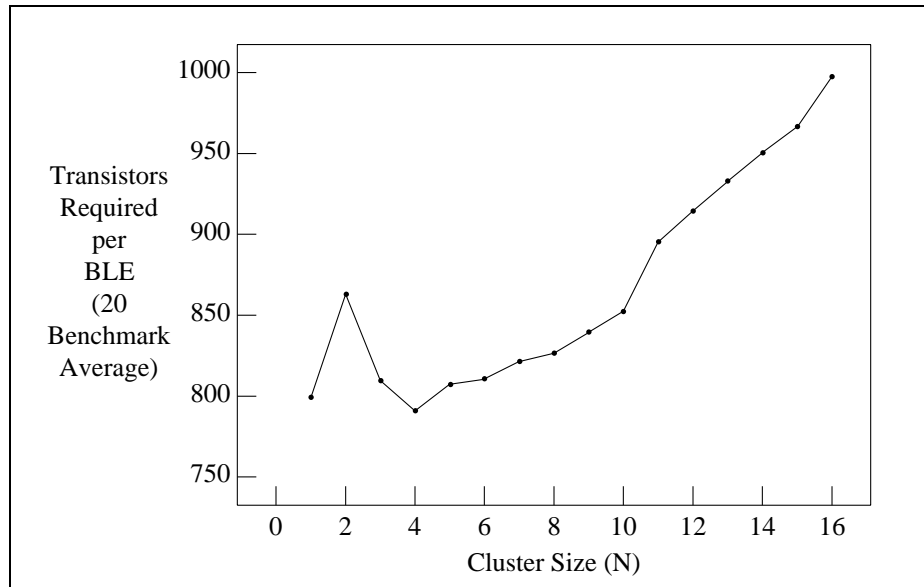


Figure 3-26 Area efficiency vs. cluster size (from Betz and Rose [Bet98], ©1998 IEEE).

LUT utilization

Figure 3-25 shows how well the lookup tables are used as a function of the number of accessible inputs. Ideally, we would like to be able to use all of the bits in the lookup tables, but the fixed interconnect in the cluster may not match well with the logic functions to be implemented. However, the figure shows that utilization reaches 100% when only 50% to 60% of the lookup table inputs are accessible; the common inputs and locally generated outputs do not cause a problem.

Betz and Rose also studied the number of routing tracks to which each pin should connect. When a cluster has a large number of pins, it is not necessary that all the pins be able to connect to all of the tracks. They found that an organization that allows each of the routing tracks to be driven by one output pin on each logic block was sufficient to ensure high utilization.

area efficiency vs. cluster size

Figure 3-26 shows how area efficiency varies with cluster size. Clusters in the 1 to 8 size range showed good area efficiency.

3.7.2 Interconnect Architecture

<i>interconnect channel considerations</i>	The design of the interconnect architecture is distinct from the design of the interconnect circuits. The circuit designs for drivers and amplifiers that we discussed in the last section are for a single wire; the interconnect architecture, in contrast, describes the entire set of wires that are used to make all the connections in the FPGA.
<i>interconnect structures</i>	<p>The connection between two LEs must in general be built through three types of connections:</p> <ul style="list-style-type: none"> • the logic element must connect to the wiring channel; • the wire segments in the wiring channel must be connected together to reach an intermediate point or destination; • a connection must be made between wiring channels. <p>The connections between wire segments in a channel can generally be made to adjacent wires. The connections into the wiring channel or between wiring channels should be richer but that richness comes at the cost of additional programming circuitry. The logic elements in an anti-fuse-based FPGA are generally connected to one wire segment in the channel. SRAM-based FPGAs generally allow an input or output of a logic element to connect to any of several wires in the wiring channel. Connections between wiring channels would ideally be crossbars that allow any wire in one channel to be connected to any wire in the other channel. However, few FPGAs offer such rich inter-channel interconnections due to the cost of the programming circuitry.</p>
<i>segmented wiring</i>	As described in Section 3.2, one of the key questions in the design of interconnect is the length of the routing segment in a general-purpose interconnection network. An individual segment may run for several LEs before it stops at a programmable interconnection point. The segment will not be able to connect to some LEs but it will provide less delay than a segment that is broken at every LE by an interconnection point. Segmented routing channels may also use offset wiring segments so that not all logic elements connect to the same wire segments.
<i>routing segment length vs. delay</i>	Brown et al. [Bro96] studied the effects of routing segment length on delay. Figure 3-27 shows routing delays for a number of different fabrics that have different proportions of segment lengths. In this graph, the vertical axis shows the percentage of tracks that were of length 2 while the horizontal axis shows the percentage of tracks that were of length 3; for data points where the sum of length 2 and length 3 tracks was less than 100%, the remaining tracks were of length 1. The figure shows that

100	8.3												
90	8.4	8.2											
80	8.4	8.1	8.0										
70	8.6	8.3	8.0	7.9									
60	9.0	8.5	8.2	7.9	7.7								
50	9.5	8.9	8.4	7.9	7.7	7.6							
40	10.3	9.5	8.7	8.2	7.9	7.5	7.5						
30	11.1	10.1	9.4	8.7	8.2	7.7	7.5	7.4					
20	11.9	11.0	10.2	9.3	8.6	8.0	7.7	7.4	7.5				
10	12.4	11.7	10.9	10.1	9.2	8.3	7.9	7.6	7.4	7.5			
0	12.8	12.5	11.8	10.8	9.9	9.0	8.4	7.9	7.5	7.4	7.5		
	0	10	20	30	40	50	60	70	80	90	100		

Figure 3-27 Routing delays of segmentation schemes (from Brown et al. [Bro96], © 1996 IEEE).

the sweet spot in the design space is centered around a point where more of the tracks are of length 3 and most of the remainder are of length 2.

Betz and Rose [Bet98] studied FPGAs with varying routing channel widths. Their study was motivated by commercial FPGAs which had larger routing channels with more wires in the center of the FPGA and smaller routing channels with fewer wires toward the edge of the chip. They found that, in fact, using the same size routing channels throughout the chip made the FPGA more routable, although if the design had the positions of its I/O positions constrained by external requirements, making the routing channels that feed the I/O pins 25% larger aided routability.

3.7.3 Pinout

Another concern is how many pins to provide for the FPGA. In a custom design the number of pins is determined by the application. But since an FPGA provides uncommitted logic we must find some other way to provide the right number of pins. If we provide too many pins we will drive up the cost of the chip unnecessarily (the package is often more expen-

sive than the chip itself). If we don't provide enough pins then we may not be able to use all the logic in the FPGA.

Rent's Rule

The best characterization of the relationship between logic and pins was provided by E. F. Rent of IBM in 1960. He gathered data from several designs and plotted the number of pins versus the number of components. He showed that the data fit a straight line on a log-log plot. This gives the relationship known as **Rent's Rule**:

$$N_p = K_p N_g^\beta \quad \text{(EQ 3-1)}$$

where N_p is the number of pins and N_g is the number of logic gates. The formula includes two constants: β is Rent's constant while K_p is a proportionality constant. These parameters must be determined empirically by measuring sample designs. The parameters vary somewhat depending on the type of system being designed. For example, Rent measured the parameters on early IBM mainframes as $\beta = 0.6$ and $K_p = 2.5$; others have measured the parameters for modern microprocessors as $\beta = 0.45$ and $K_p = 0.82$.

3.8 Summary

FPGA fabrics are complex so that they can support realistic system designs. They contain several different components: logic elements, multiple types of interconnection networks, and I/O elements. The characteristics of FPGA fabrics are determined in part by VLSI technology and in part by the applications for which we want to use FPGAs.

3.9 Problems

Q3-1. You have a two-input lookup table with inputs a and b. Write the lookup table contents for these Boolean functions:

- a. a AND b.
- b. NOT a.
- c. a XOR b.

Q3-2. You have a three-input lookup table with inputs a, b, and c. Write the lookup table contents for these Boolean functions:

- a. a AND b.

- b. $a \text{ AND } b \text{ AND } c$.
- c. $a \text{ XOR } b \text{ XOR } c$.
- d. $a + b + c$ (arithmetic).

Q3-3. You have a logic element with two lookup tables, each with three inputs. The output of the first lookup table in the pair can be connected to the first input of the second lookup table using an extra configuration bit. Show how to program this logic element to perform:

- a. $a + b + c$ (arithmetic, sum only not carry).
- b. $a - b$ (arithmetic, difference only not borrow).

Q3-4. Redesign the logic element of Figure 3-11 to be controlled by a_0 OR a_1 in the first stage and b_0 AND b_1 on the second stage. Draw the schematic and write the truth table.

Q3-5. Design each of these functions using a tree of multiplexers:

- a. $a | \sim b$.
- b. $a \& (b | c)$.
- c. $(a \& \sim b) | (c \& d)$.

Q3-6. Program the logic element of Figure 3-13 to perform these functions:

- a. $a \& b$.
- b. $a | b$.
- c. $a \text{ NOR } b$.
- d. $ab + bc + ac$.

Q3-7. How many two-input LUTs would be required to implement a four-bit ripple-carry adder? How many three-input LUTs? How many four-input LUTs?

Q3-8. Prove that the fast arithmetic circuitry of the Actel Axcelerator performs a correct two-bit addition with carry.

Q3-9. Draw a transistor-level schematic diagram for the programmable interconnection point shown in Figure 3-6. The interconnection point should be controlled by a five-transistor SRAM cell.

Q3-10. Populate the array of logic elements in Figure 3-2 with wires and programmable interconnection points. Each wiring channel should have two wires. Assume that each logic element has two inputs and one output; each logic element should be able to connect its inputs to the channel on its left and its output to the channel on its right. When two wiring

channels cross, you should be able to make connections between all the crossing wires.

Q3-11. Redo your routing design of Question Q3-10 but add local connection wires. Each local wire should be the output of an LE to one of the inputs of the LE on its right. Each end of the local connection should be controlled by a programmable interconnection point.

Q3-12. Your FPGA has 128 logic elements, each with four inputs and one output. There are 128 vertical and 128 horizontal routing channels with four wires per channel. Each wire in the routing channel can be connected to every input of the LE on its right and the output of the LE on its left. When two routing channels intersect, all possible connections between the intersecting wires can be made. How many configuration bits are required for this FPGA?

Q3-13. Draw a transistor schematic for two programmable interconnection points that are connected in a scan chain. Each programmable interconnection point should be implemented by a five-transistor SRAM cell.

Q3-14. Draw a schematic for a four-input multiplexer that uses a combination of pass transistors and static gates. The first stage of multiplexing should be performed by pass transistors while the remaining multiplexing should be performed by static gates.

Q3-15. Draw a transistor-level schematic for a programmable interconnection point implemented using a three-state buffer.

Q3-16. Draw eight LEs and a routing channel with eight wires. The routing channel should have two sets of length 1 segments, two sets of length 2 segments, and four sets of length 3 segments. Each LE should be able to be connected to at least one length 2 and one length 3 segment.

Q3-17. Draw a block diagram for a logic element cluster with two two-input LEs and four inputs to the cluster. The local interconnect network in the cluster should be able to connect the two inputs of an LE to two of the cluster inputs.