2

Reconfigurable Logic Devices

Though Gerald Estrin [139, 141, 142] had conceived of reconfigurable computing as early as 1960, the relatively recent developments in reconfigurable computing have been fueled by the availability of logic devices that can be quickly and easily programmed and reprogrammed to perform a large variety of functions. The first devices of this type that achieved enough density to perform significant portions of a computation and that had significant availability were field-programmable gate arrays (FPGAs). These chips provide the designer with arrays of simple logic functions and memories (such as flip-flops) that can be connected through programmable interconnection networks. To begin with, the early FPGA devices from Xilinx, Altera, and others provided relatively little logic, but later generations provided enough logic for researchers to consider what might be possible through direct implementation of computational algorithms in reconfigurable logic devices. The densities of today's FPGAs have exceeded 150,000 4-input look-up tables (LUTs) per device and some have developed into devices that can be used to build complete systems on a programmable chip (SoPC), providing such specialized features as digital signal processing (DSP) blocks, multi-gigabit serial I/O, embedded microprocessors, and embedded SRAM blocks of various sizes.

In addition to the relatively fine-grained configurability provided by FPGAs and similar devices, the drive to reduce the power, area, and/or delay costs of fine-grained reconfigurability has led to a number of what may be called "coarse-grained" reconfigurable logic devices. Instead of providing configurability at the level of individual gates, flip-flops or look-up tables (LUTs), these coarse-grained architectures often provide arithmetic logic units (ALUs) and other larger functions that can be combined to perform computations. In the extreme, the functions might be as large as microprocessor cores such as in the Raw chip [408] from MIT.

The goal of this chapter is to provide the reader with a brief overview of what reconfigurable logic is and some of the important forms of reconfigurable logic that have been used in reconfigurable computing. This chapter will discuss, in general terms, the features of both fine- and coarse-grained

reconfigurable logic devices and will describe some significant examples of each.

2.1 Field-Programmable Gate Arrays

With their introduction in 1985, field-programmable gate arrays (FPGAs) have been an alternative for implementing digital logic in systems. To begin with, FPGAs were used to provide a denser solution for glue logic within systems, but now they have expanded their applications to the point that it is not uncommon to find FPGAs as the central processing devices within systems. Compared with application-specific integrated circuits (ASICs) and mask-programmable gate arrays (MPGAs), FPGAs have several advantages for their users, including: quick time to market, being a standard product; no non-recurring engineering costs for fabrication; pre-tested silicon for use by the designer; and reprogrammability, allowing designers to upgrade or change logic through in-system programming. By reconfiguring the device with a new circuit, design errors can be fixed, new features can be added, or the function of the hardware can be entirely retargeted to other applications. Of course, compared with ASICs and MPGAs, FPGAs cost more per chip to perform a particular function so they are not good for extremely high volumes. Also, an FPGA implementation of a function is slower than the fixed-silicon options. Over time, though, the expense of doing custom silicon and the fact that FPGAs now tend to use state-of-the-art CMOS processes mean that FPGAs are performing more of the functions that ASICs and MPGAs would have performed in many systems.

Considering the complexity of contemporary FPGA devices and the several existing texts describing FPGA architecture and design [50, 59, 395], this section will provide only a brief overview of FPGA architecture and features as a foundation for later discussions about how FPGAs are used in reconfigurable computing. This overview will first discuss basic FPGA architecture, including basic logic, routing, and input/output (I/O) structures. Then, we will introduce some of the more advanced features that have been integrated into FPGAs since the late 1990's, including embedded memory, embedded arithmetic logic, high-speed serial I/O, and embedded microprocessors. While introduced from the perspective of FPGAs in this chapter, tightly integrated reconfigurable logic and microprocessors will be discussed later in Chapter 3 in the context of reconfigurable computing systems. This section on FPGAs will conclude with a discussion on FPGA programming architecture.

2.1.1 Basic Architecture

The basic architecture of FPGAs consists of three kinds of components: logic blocks, routing, and input/output blocks. Generally, FPGAs consist of an array of programmable logic blocks that can be interconnected to each other

as well as to the programmable I/O blocks through some sort of programmable routing architecture. Figure 2.1 provides a very simplified diagram of a generic FPGA architecture.

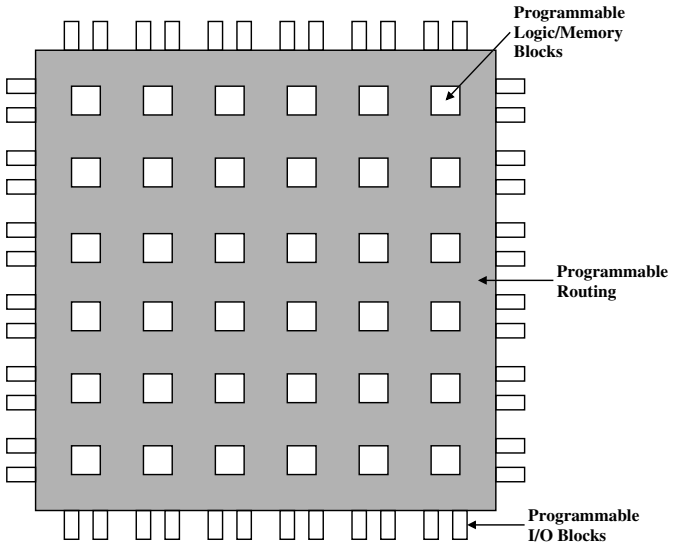

Fig. 2.1. A Generic FPGA Architecture

With fine-grained architectures such as FPGAs, the logic operations are mainly done at the bit or small word (≤ 4 bits) level, providing a very flexible architecture that can be customized to the specific needs of an application. For instance, if an application only needs a 6-bit adder for a particular operation and a 23-bit adder for another, the designer does not need to use a 32-bit adder to perform these lower precision computations—the designer can directly implement the size of adder that is needed for the specific application. This flexibility does comes at a significant cost compared with a non-programmable, custom silicon implementation due to the large number of transistors and the large amount of wiring needed to provide the fine-grained programmability. Numbers such as 10X-100X cost in terms of silicon area and 10-100X cost in circuit speed are frequently quoted [8] for FPGA implementations as compared to custom ASICs. Designers of FPGA chips try to balance the costs of flexibility with those of chip size and speed to give FPGA users both the flexibility they desire while reducing chip costs and increasing chip speeds. The next several paragraphs will describe the main three elements of FPGAs and some typical examples of how they are constructed.

Programmable Logic

FPGA designers have developed a large variety of programmable logic structures for FPGAs since their invention in the mid-1980's. For more than a decade, much of the programmable logic used in FPGAs can be generalized as shown in Figure 2.2. The basic logic element generally contains some form of programmable combinational logic, a flip-flop or latch, and some fast carry logic to reduce the area and delay costs for implementing carry logic. In our generic logic block, the output of the block is selectable between the output of the combinational logic or the output of the flip-flop. The figure also illustrates that some form of programming, or *configuration*, memory is used to control the output multiplexer; of course, configuration memory is used throughout the logic block to control the specific function of each element within the block.

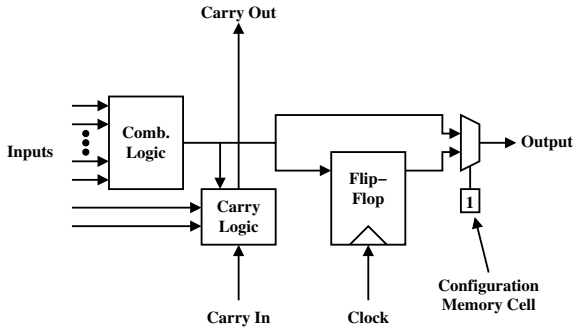

Fig. 2.2. A Generic Programmable Logic Block

Unlike our generic logic element, commercial FPGA devices generally provide a large amount of flexibility within the logic element. For instance, a flip-flop in many commercial FPGAs can be made to operate as a simple latch, can be programmed to have several combinations of asynchronous or synchronous sets and resets, and can be negative- or positive-edge triggered. In recent Xilinx and Altera devices, the carry logic has evolved into logic that supports additional functions. For instance, with the Xilinx Virtex FPGA, the carry logic has been augmented to help with multiplication. In the latest Stratix II FPGA from Altera [260], full-adders have taken the place of carry logic.

Regarding the combinational logic portion of the logic element, many different implementation methods have been used. The most common way to implement the combinational logic is with a look-up table (LUT). Figure 2.3 illustrates how a three-input look-up table is conceptually implemented—a series of programmable memory cells with a multiplexer to select the output of a specific memory cell. The look-up table, of course, operates as a memory

with N address lines and 2^N memory locations. To implement a specific logic function with the memory, the truth table for the function is loaded into the memory. For example, the look-up table in Figure 2.3 implements an three-input AND function. Note that the LUT only produces a logic "1" when all address lines (labeled A, B, and C) are a logic "1"; this, of course, is due to the fact the memory cell at address 7 is the only cell storing a "1" value. Due to area efficiency [348], most commercial SRAM-based FPGAs use four-input LUTs for their combinational logic elements.

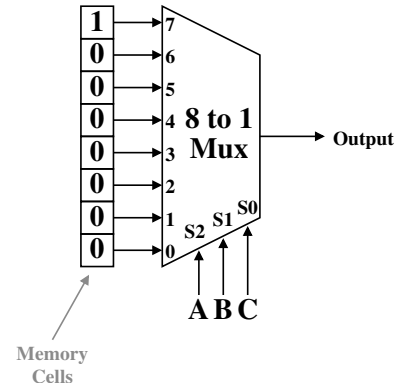

Fig. 2.3. A Three-Input Look-Up Table

Though most reprogrammable FPGAs use LUTs for combinational logic, several architectures (e.g., [5,162,234]) have used combinations of multiplexers and logic gates to implement programmable logic structures. An example of this approach can be seen in the Actel ProASIC Plus logic element shown in Figure 2.4, which can produce all three-input/one-output functions except for the three-input exclusive OR (XOR) and can even operate as a master-slave flip-flop of various types. Reprogrammable FPGAs using these alternative combinational logic architectures tend to implement functions of a finer granularity (2- and 3-input functions are common) at the logic-element level than LUT-based architectures.

To reduce the costs of using programmable routing, many reprogrammable FPGA architectures will cluster the logic elements together using fast, short-length routing. This clustering allows larger functions to be created using only the faster routing of the cluster. Most recent LUT-based architectures employ this strategy, often pairing two or more four-input LUT-based logic elements into a cluster. Over time, due to the delay costs of general-purpose programmable routing, commercial LUT-based FPGAs have increased the size of the clusters to improve circuit performance. Table 2.1 illustrates this increased clustering over time. In their Stratix II FPGA [15,260], Altera uses a mini-cluster they call the "Adaptive Logic Module" (ALM) consisting of four

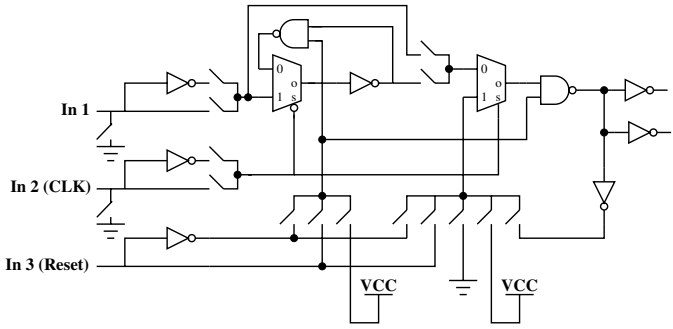

Fig. 2.4. Actel ProASIC Plus Logic Element

three-input LUTs and two four-input LUTs to create a flexible combinational logic structure that can handle up to a single function of 7 inputs or two independent functions of between 3 to 5 inputs. The full cluster in Stratix II, called a Logic Array Block (LAB), then collects 8 ALMs into a larger unit, resulting in a cluster of 24 three-input LUTs and 16 four-input LUTs.

Device Name	Year	LUT Width	Cluster Name	Cluster Size
Xilinx XC2000	1985	4	CLB	1
Xilinx XC3000	1987	4	CLB	2
Xilinx XC4000	1990	3 & 4	CLB	1 (3LUT) & 2 (4LUT)
Altera FLEX 8000	1992	4	LAB	8
Altera FLEX 10K	1995	4	LAB	8
Xilinx Virtex	1998	4	CLB	4
Altera Apex 20K	1998	4	LAB	10
Xilinx Virtex-II	2000	4	CLB	8
Altera Apex II	2001	4	LAB	10
Altera Stratix	2002	4	LAB	10
Xilinx Virtex-4	2004	4	CLB	8
Altera Stratix II	2004	3 & 4	LAB	24 (3LUT) & 16 (4LUT)

Table 2.1. Logic Element Cluster Sizes of LUT-Based FPGAs over Time

Routing

As with logic element structures, FPGA designers have used a variety of routing structures within their FPGAs. Various forms of routing exist through out each FPGA architecture. Generally, some amount of routing is included within each logic cluster so that the logic elements can be combined to form larger functions. External to the logic clusters is the more global routing architecture of the FPGA. In this section we will concentrate mainly on global

routing architectures, though some general comments will be made about the internal cluster routing.

To implement programmable routing, three basic switch types are used: multiplexers, pass transistors, and tri-state buffers. Figure 2.5 illustrates each of these switches with an SRAM cell controlling their outputs. Generally, pass transistors and multiplexers are used within a logic cluster to connect the logic elements together while all three are used for the more global routing structure. Multiplexers are very common switch elements in FPGAs and they come in a large variety of widths ranging from two inputs to eight or more inputs, depending on the complexity of the routing network. Every time a signal is routed through pass-transistor-based switches, more capacitance is added to the load of the driving transistors. To counteract the resulting slow down due to capacitance, most modern FPGAs include active buffering in the routing networks.

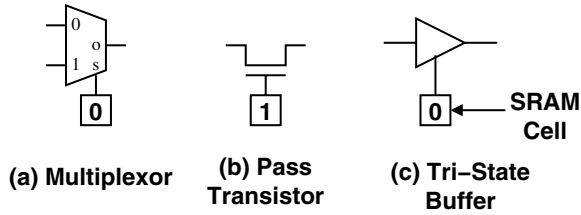

Fig. 2.5. Basic Programmable Switch Types

Within a logic cluster, routing is used for several purposes. First it is used to determine where the inputs to the logic elements come from and where the outputs will go. Next, programmable routing is used to determine how signals propagate through the logic elements themselves. Further, non-programmable routing is generally used for fast carry propagation to eliminate the extra delays incurred when using programmable routing. Usually, these carry chain paths extend between logic clusters as well, to support wide additions. Finally, routing within the logic cluster is frequently used to combine the logic elements into wider or otherwise more complex functions. Figure 2.6 illustrates different sorts of routing within a logic cluster.

Several global routing architectures have been implemented in FPGAs, the main four can be categorized as the island, cellular, long-line, and row architectures [49,395]. We will briefly describe each. Note that modern FPGAs have routing architectures which are significantly more complex than what we describe, but these general architectures are still identifiable within these FPGAs.

Figure 2.7 illustrates the basic island-style routing architecture. In this routing architecture, logic clusters are surrounded by segmented horizontal and vertical routing channels. Each cluster connects to the routing through "connection boxes" and each segment in the routing can be connected to an-

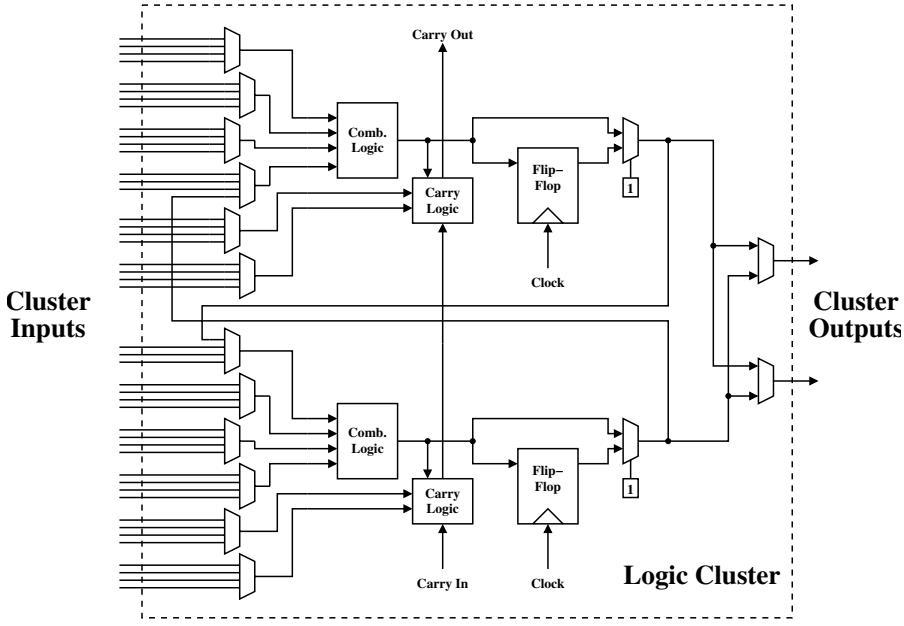

Fig. 2.6. Internal Logic Cluster Routing

other segment through a "switch box." The main feature of this type of routing is that connections between logic clusters are made through segmented routing. This architecture is found in many Xilinx FPGA architectures, though, Xilinx provides segments in several lengths (even wires that span the entire chip) while also providing local routing between logic clusters to make the architecture more efficient.

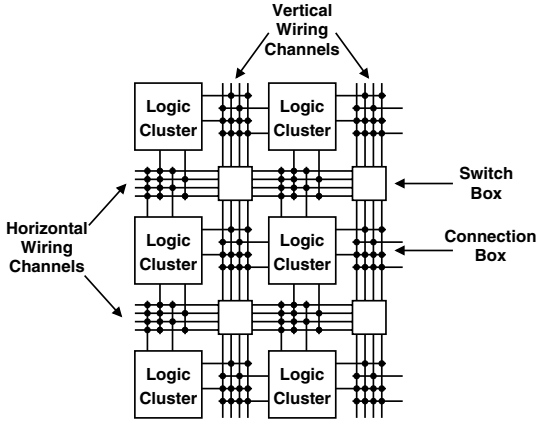

Fig. 2.7. Island Routing Architecture

The next architecture, the long-line routing architecture, takes a different approach. As illustrated in Figure 2.8, the long-line architecture also surrounds logic clusters with horizontal and vertical routing channels with multiple wires per channel, but each of the wires spans the width or height of the entire chip. Ideally, to connect any two logic clusters in this arrangement, only one vertical and one horizontal long line is required. The transition between a long vertical line and a long horizontal line can be made by using the internal routing of a logic cluster at the intersection of the two lines. This has been the main routing architecture for Altera FPGAs, but other FPGAs such as Actel's ProASIC FPGAs have similar routing structures. As illustrated in the figure, Altera's routing architecture generally provides for horizontal, local inter-cluster routing as well. To reduce the speed penalty for driving wires that span the length of the chip, several of Altera's latest architectures, such as Stratix [261] and Stratix II [15, 260], have introduced smaller length segments in the routing channels.

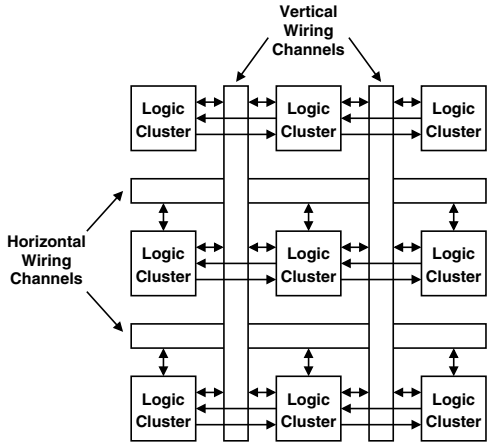

Fig. 2.8. Long-Line Routing Architecture

The cellular routing architecture, shown in Figure 2.9 is different yet from the island and long-line architectures. In this routing architecture, the richest connections are made locally between logic clusters and only a small amount (if any) are made through longer wire segments. Examples of FPGAs with this type of routing structure include the Algotronix CAL FPGA [234] (which later became the Xilinx XC6200), the CLi/Atmel 6000 FPGAs [162], and the Plessey/Pilkington ERA [215]. For the most part, these architectures tend to be very fine grained (2 or 3 input functions), so logic clusters are really very simple and have a single logic element in them. To aid with routing, the logic cells themselves were designed so they could be used as a part of the routing

network between other logic elements. The cellular style of routing has fallen out of favor for several reasons:

- the delays for combinational paths can be significant for circuits that require more than nearest neighbor routing;
- CAD tools seemed to have significant difficulty efficiently mapping and routing circuits to the architectures (though, this may be true, in part, due to a lack of investment in the tools); and
- the area and delay costs of fine-grained architectures and routing are significant due to the amount of programmable logic and routing required to implement a function.

The final point can be mitigated somewhat if the design can be heavily pipelined since each cell has a flip-flop, but few designs come maximally pipelined. Recently, Cell Matrix [131], an FPGA-like architecture proposed for fabrication with nano-scale technologies, has revived interest in this routing architecture partially because of its relative simplicity to fabricate due to regularity.

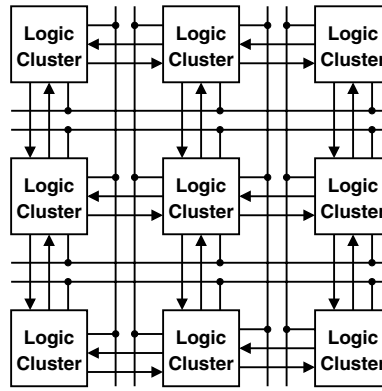

Fig. 2.9. Cellular Routing Architecture

Finally, for completeness, the row architecture for routing is illustrated in Figure 2.10. This routing architecture is mainly found in one-time programmable FPGAs such as many of Actel's anti-fuse FPGAs and is, therefore, not commonly seen in FPGAs used for reconfigurable computing. The architecture chiefly uses horizontal interconnect channels to route signals between two logic clusters. As suggested in the figure, Actel FPGAs (and others) do generally provide some vertical routing despite the bias toward row-based routing channels. For instance, in the Act-1 and later FPGAs, Actel used vertical wires to route the outputs of the logic clusters to several adjacent routing channels and provided some long wires that spanned even more row routing channels. Though not illustrated in the figure, row-based routing architectures

generally used segmented wires within the routing channels to reduce routing delays for short paths.

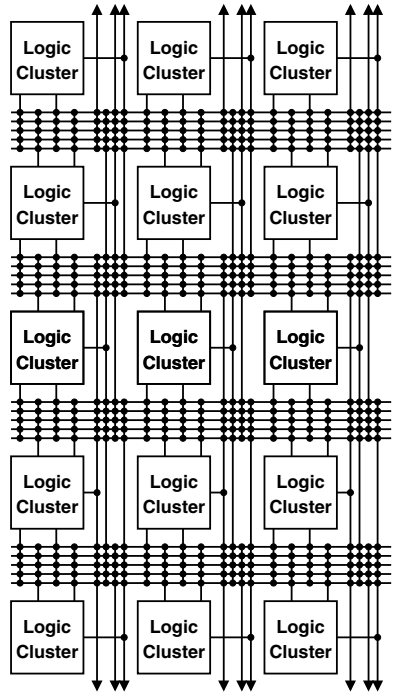

Fig. 2.10. Row Routing Architecture

Programmable I/O Architectures

Unlike logic and routing architectures, the basic input/output (I/O) architecture, as shown in Figure 2.11, is reasonably similar across FPGA families. In effect, the I/O blocks have tri-state buffers for the outputs and input buffers for the inputs. The tri-state enable signal, the output signal, and the input signals can be individually registered within the I/O block or can be left unregistered based on how the I/O block is programmed.

In modern FPGAs, a wide variety of additional features can be found that significantly enhance and complicate this basic structure. For instance, the Xilinx Virtex-4 architecture's I/O blocks [427] provide the following features, among others:

- more than 50 variations of I/O signaling standards some of which include features such as:

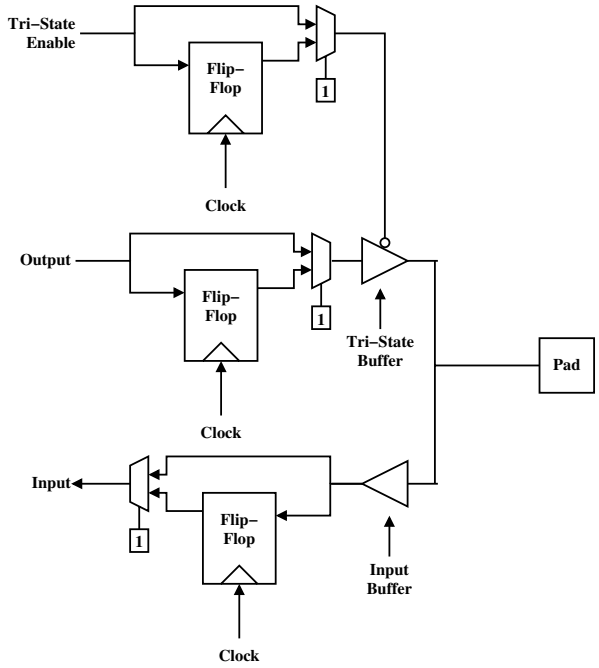

Fig. 2.11. I/O Block Architecture

  - digitally controlled impedance to eliminate the need for terminating resistors to deal with transmission line effects and
  - differential signaling to improve signal integrity;
- double-data rate registering with multiple modes for presenting and receiving data to and from the internal logic; and
- programmable input delays.

2.1.2 Specialized Function Blocks

Over time, the basic FPGA architectures that we have described above have been further developed through the addition of more specialized programmable function blocks. These blocks—such as embedded memory, arithmetic logic, high-speed serial I/O, and even embedded microprocessors—have been added due to a frequent need for such resources within FPGA systems and applications. The result is that many recent FPGAs are a more heterogeneous mixture of resources than early FPGAs. In the next few paragraphs, we will briefly describe the resources that have been made available in recent FPGAs.

Embedded Memory

Memory, of course, is a basic component of most digital systems and, though flip-flops can be used for memory, they are very inefficient for creating memories of any depth. Starting with the XC4000 series of FPGAs, Xilinx made the LUTs used for logic flexible enough to be used as asynchronous 16x1 RAMs in user designs. In later architectures, Xilinx has designed the LUTs so that they can operate as synchronous RAMs, dual-ported RAMs, and shift registers (1 to 16 stages). Logic also exists in the logic clusters to compose these smaller RAMs into wider or deeper RAMs. Being able to use LUTs for RAM is a feature unique to Xilinx FPGAs.

Though LUT-based memory is better than using flip-flops when implementing deeper memories, it was not long until most FPGA vendors started to include more dense blocks of SRAM within the architectures, with Altera leading the way with its FLEX 10K FPGAs [13]. Today, Altera, Xilinx, Actel, Atmel, Quicklogic, and other FPGA vendors include these larger SRAMs that have from hundreds to thousands of bits. Most of the RAMs tend to be on the range of 1- to 4-Kilobits (Kb), though, in the Stratix II architecture, Altera has three granularities of RAMs (576 Kb, 4.5 Kb, and .56 Kb) available to the designer. In many cases, the RAMs aspect ratio can be programmed. For instance, a Xilinx Virtex 4-Kb RAM can operate in the following possible modes: 4096x1, 2048x2, 1024x4, 512x8, 256x16 (where the aspect ratios are given as *depth* x *width* in bits). Besides having programmable aspect ratios, some FPGAs' embedded RAMs can operate as content-addressable memories (CAMs) [14], dual-ported RAMs, and/or FIFOs.

Though the total memory available on chip may only be as much as 1 MB for the largest FPGAs, one of the key benefits of these on-chip memories is the large number of memory ports available and the aggregate memory bandwidth that is possible, providing a significant advantage to very parallel applications that require significant memory bandwidth. For instance, the largest announced Stratix II (EP2S180) theoretically can provide a maximum aggregate memory bandwidth over 30 Gb/s through the 3414 ports of its 1707 RAMs (assuming each memory operates dual-ported and operates at maximum frequency) [15].

Embedded Arithmetic Logic

Beyond the basic carry logic and even adders provided in the logic elements and logic clusters, many FPGAs have started to include 18x18 multipliers or so-called "Digital Signal Processing" (DSP) blocks as separate, additional resources. In general, the DSP blocks provide addition/subtraction, multiplication, and multiply-accumulate (MAC) operations with a high degree of configurability. The MAC operations are useful in finite-impulse response (FIR) filtering, a common DSP operation (see Chapter 5 for a discussion on FIR filtering and FPGAs). A detailed description of these blocks for the latest

FPGAs from Xilinx and Altera is out of the scope of this chapter, especially, since the DSP blocks can be so flexible and complex (see [16, 429] for more detailed information), but Table 2.2 provides a summary of some of their features.

Features	Xilinx Virtex-4 XtremeDSP	Altera Stratix II DSP Block
Multiply-accumulate	48-bit	52-bit
Multiply	18x18, multi-precision (iterative)	8 9x9, 4 18x18, 1 36x36
Add/Subtract	Yes	Yes
Complex multiply support	Yes	Yes
Integrated FIR support	Yes	Yes
Saturating arithmetic	No	Yes
Routing arithmetic	Yes	Yes
Fixed point representation	No	Yes
Barrel Shifter	Yes	No
Iterative functions	Extended width multiply, Integer division, Integer square root	Possible(?)

Table 2.2. Xilinx Virtex-4 and Altera Stratix II DSP Block Features

High-Speed Serial I/O

Considering that many FPGAs are used in high-throughput telecommunications equipment, the recent addition of multi-gigabit serial transceivers (MGSTs) as I/O blocks may not be too surprising. These blocks perform full-duplex serialization and deserialization functions (SERDES), provide encoding/decoding functions (for instance, 8B/10B), and some error control logic. The first production FPGA to have this capability was Xilinx's Virtex-II Pro (up to 6.25 Gb/s full-duplex per channel). Altera's Stratix GX family (up to 6.25 Gb/s full-duplex per channel) and Xilinx's Virtex-4 FX family (up to 11.1 Gb/s full-duplex per channel) of FPGAs also provide this capability. Taking the concept of high speed communication even further, the Virtex-4 FX family of FPGAs has two or four dedicated 10/100/1000 Mb/s Ethernet Media Access Controllers (MACs) to provide designers with a more complete solution for Ethernet serial I/O solutions.

Embedded Microprocessors

Lastly, FPGA manufacturers have integrated full, dedicated microprocessors with their FPGA logic to perform low-bandwidth and/or control-intensive

functions such as implementing TCP/IP stacks. With this capability, complete (or near complete) embedded systems can be implemented with a single device. The first commercial FPGA to integrate a microprocessor with its logic fabric was the Altera Excalibur. When announced the Excalibur was the integration of either an ARM- or MIPS-based 32-bit RISC processor core with an APEX-20KE-like logic fabric. Currently, only the ARM-based solution is available from Altera. As illustrated in Figure 2.12, the processor has dedicated external memory interfaces, a UART interface, a programmable interrupt controller, and other resources. Two dedicated AMBA high-performance buses (AHBs) exist in the system, providing the processor access to two different tiers of devices. The processor communicates with the programmable logic through the secondary AHB or through some dual-port SRAM that connects to both buses.

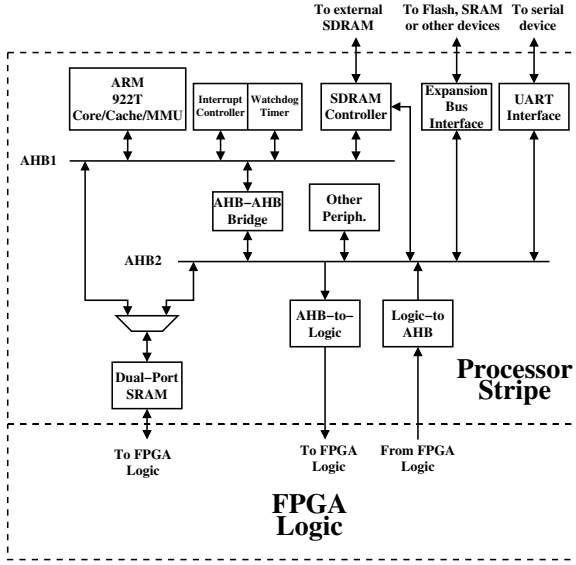

Fig. 2.12. ARM-based Excalibur Architecture

Later, with the Virtex-II Pro and Virtex-4 FPGAs, Xilinx produced FPGAs with integrated integer PowerPC microprocessors. One significant difference between the Xilinx approach to processor-logic integration and that of Altera is that the Xilinx approach places the microprocessor as an island within the FPGA logic, as illustrated in Figure 2.13, with interfaces to on-chip SRAM but no dedicated processor or peripheral buses. These buses must be implemented using FPGA logic, if they are desired. This provides the designer with the flexibility to define the architecture of the embedded system, but also means that the processor cannot perform useful work without configuring the FPGA logic, unlike the situation with the Excalibur. Note that the processors

in Xilinx FPGAs replace any logic functions that would have normally occupied that area of the chip, but some of the routing architecture is maintained despite the presence of the PowerPC.

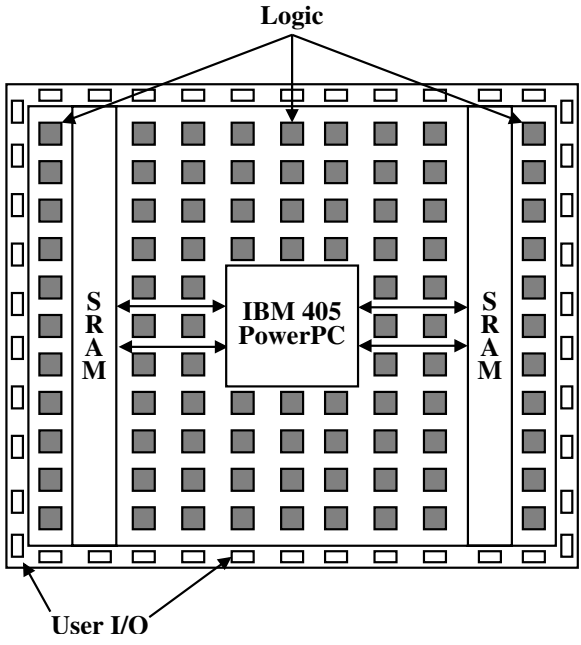

Fig. 2.13. Xilinx-Style Processor Integration

2.1.3 Programming Architecture

Reprogrammable FPGAs generally use SRAM to store the configuration data. The programming architecture of SRAM-based FPGAs is an important factor in the use of these FPGAs for reconfigurable computing. By programming architecture, we mean the way in which the FPGA's configuration data is structured and how it is provided to the FPGA to program its resources.

Many approaches to programming architecture have been developed for FPGAs, but the main characteristics of interest include the programming bandwidth, the granularity of the accesses, on-line programmability, and the ability to read out the programming data. We will discuss each briefly and describe their impact on reconfigurable computing.

Programming bandwidth, as you might guess, is the rate at which configuration data can be sent to the FPGA. When fast reconfiguration of an FPGA is required—such as when an application is time multiplexing logic on the hardware, the programming bandwidth is one factor that determines

how fast the FPGA can be reprogrammed. The width of the programming interfaces are generally either serial or 8-bit interfaces and can run at tens of MHz. The Altera Stratix II FPGA, for example, can obtain a bandwidth of about 800 Mb/s for configuration in the Fast Passive Parallel mode—an 8-bit interface running at 100 MHz. As another example, the Xilinx Virtex-II Pro can operate up to 400 Mb/s. A common serial interface is the Joint Test Action Group's boundary scan interface (IEEE 1149.1)—which is often referred to as simply JTAG—that has been adapted for configuration. With JTAG, Altera's Stratix II FPGA achieves only about 10 Mb/s, where many Xilinx devices can have a bandwidth of up to 33 Mb/s. Some FPGAs have proprietary serial programming interfaces as well with higher data rates.

The granularity of the FPGA's programmability determines how many resources are configured with the smallest block of programming data. As far as the FPGA chip designer is concerned, this granularity has a significant effect on the cost of the internal logic used to perform the programming. Like with other forms of memory, the more addressability that is required within the programming data, the more logic that is needed to provide that addressability.

With respect to reconfigurable computing, the main interest in the granularity of the FPGA programming data is related to another aspect of an FPGA's programming architecture—the support provided by some FPGAs to perform on-line programming (or dynamic configuration) of a portion of their logic. This allows the FPGA user to modify the circuit as it executes. The programming granularity, then, has an impact on the smallest amount of data and logic that can and must be affected when making small changes to a circuit (for instance, reloading the ROMs holding coefficients used in some arithmetic or DSP algorithm).

As an example, the Xilinx Virtex series of FPGAs (Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-4) allow partial reconfiguration of the FPGA. The smallest amount of data that can be programmed is called a frame, which is typically hundreds to thousands of bits. So, to partially reconfigure the FPGA logic at run time (sometimes referred to as run-time reconfiguration [417]), an entire frame of configuration data must be rewritten to the FPGA for the smallest change. By contrast, the now unavailable Xilinx XC6200 [81] (a later commercial version of the Algotronix CAL FPGAs [234]) allowed the user to send as few as 8, 16, or 32 bits per configuration operation and a mask could be used to restrict the modifications of a programming data write to a specific bit or set of bits.

The ability to read an FPGA's programming data and other information through the configuration interface—a capability sometimes referred to as "readback"—has also had an impact on how FPGAs are used in reconfigurable computing. First, it can be used to ensure that the programming data stored in the FPGA is correct—a concern when using FPGAs in space [161] or even in large quantities on the ground [151]. Next, it can be used as a way of sampling the state of a user's design, as with the Xilinx Virtex series of FPGAs as well

as other such as the Xilinx XC6200. For instance, with the Xilinx Virtex FPGA, the outputs of the slices and the outputs of the IOBs (to the outside world and to the rest of the FPGA) can be sampled and read out through the FPGA's configuration interface along with the contents of the various RAMs. This can be used as a means of communication between an FPGA design and an external host (e.g., [63]) and it has been used for helping designers debug reconfigurable computing applications [181, 182, 205].

In addition to above characteristics, several modern FPGAs include programming data compression and programming data encryption capabilities. Compression, of course, reduces the amount of data that must be sent to the FPGA to program it, thus, reducing the storage requirements of the programming data and reducing programming time. The encryption capabilities have been added to FPGAs to prevent people from stealing configuration bitstreams from an FPGA system as it is being sent to the FPGA, say over the Internet or on the same system board. Though people can read the encrypted configuration data freely, a decryption key is stored on the FPGA (and kept available through battery power) before sending the system to the customer so the FPGA can decrypt the bitstream without allowing others to easily steal it. Readback of the configuration data is disabled when the configuration bitstream is encrypted to prevent unauthorized access to the actual programming data. Bitstream encryption is one of several components necessary for secure programming of reconfigurable computing machines through networks such as the Internet.

As a final note on programming architecture, several research FPGA devices [109, 265, 355, 396] have extended configuration in a novel way. These FPGAs support switching among multiple configuration states stored on the FPGA . This allows the FPGA circuit to change rapidly while the data in the FPGA either remains in place or is itself swapped in and out of the array, thus, emulating a much larger FPGA. Once the multiple, on-chip configuration memories are loaded, this considerably reduces the external programming bandwidth required to context switch among FPGA configurations since switching between device contexts (i.e., individual device configurations) is all done through accesses to on-chip configuration memory.

2.2 Coarse-Grained Reconfigurable Arrays

The configurability of fine-grained reconfigurable logic devices allows designers to specialize their hardware down to the bit level, meaning that, if an application requires 7- or 17-bit arithmetic for an operation, the hardware can directly accommodate what is needed. The configurability makes devices, like FPGAs, suitable to implement a large variety of functions directly. This configurability comes at a significant cost in terms of circuit area, power, and speed. Every level of configurability requires more multiplexing, buffering, routing, and/or memory, thus, requiring more transistors and their interconnection.

In recognizing these costs, many researchers [11,51,69,72,99,174,198,201, 256,280,291,294,380,408,439] have studied how to use arrays of more coarse-grained reconfigurable operators as the basis for reconfigurable computing machines. Besides the potential advantages in terms of circuit area, power, and speed, many researchers have pursued the use of coarse-grained reconfigurable arrays (CGRAs) with the hope that they would be easier targets for higher level development tools. However, as pointed out in Hartenstein's brief overview of 19 different coarse-grained architectures [197], the challenge with using CGRAs is that there is no universal form that is effective for all applications. To be more efficient, CGRAs are optimized in terms of operators and routing for some specific problem domain or set of domains. Thus, an application must be well matched with the array to realize dramatic improvements over other reconfigurable solutions. For example, an application that performs many 8-bit operations wastes a significant amount of logic if the CGRA uses 32-bit ALUs and operators. Considering the large number of CGRAs, we will briefly describe just a few notable research architectures as well as a few recent commercial architectures to illustrate some interesting past and current examples of CGRAs.

2.2.1 Raw

The first coarse-grained architecture we will briefly describe is probably one of the most coarse-grained—the Raw chip [386,408] from MIT. Effectively, it is a two-dimensional array of programmable tiles, each having: a 32-bit MIPS-like microprocessor, local instruction and data caches, and a 32-bit pipelined floating point unit (FPU) as well as several routers and wiring channels to support the four on-chip 2-D mesh networks. At one point, the designers had considered putting reconfigurable logic into each of the tiles, but this option was later abandoned. Figure 2.14 roughly illustrates a 4x4 Raw array—the array size that has actually been fabricated [386].

Unlike a traditional bus-based multiprocessor system, the Raw machine uses a switched network for communicating directly between the processors. The length of wires in the architecture are bounded by the width of a tile and each routing segment is registered on tile boundaries. Thus, to ensure that the array can scale, the width and height of tiles are limited by the distance a signal can travel in a single clock cycle. Two of the mesh networks are statically scheduled for predictable performance while the other two allow for dynamic scheduling through wormhole routing. Programmable routers handle the flow of data through the system and FIFOs help synchronize the transfer of data between tiles. The statically scheduled networks provide the best performance and significantly lower latency and lower overhead than traditional multiprocessor communication, enabling very fine grained coordination of operations between processors. For instance, an operand produced by one tile can be processed the following cycle by its neighbor, enabling the RAW machine to operate as a pipeline of ALUs or FPUs for a stream of data and not just

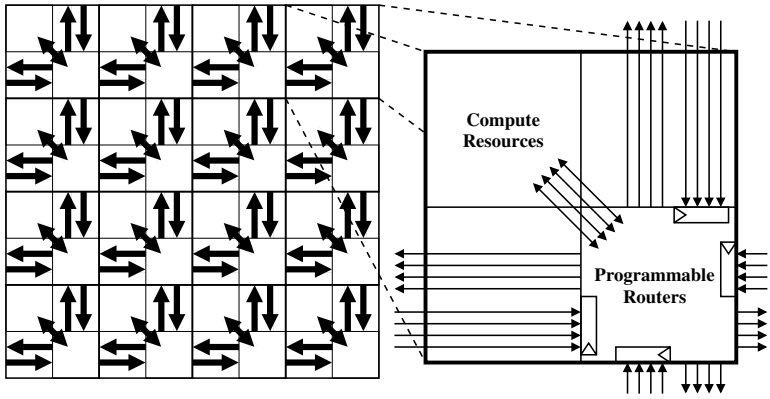

Fig. 2.14. Raw Microprocessor Array Architecture

a traditional multiprocessor system on a chip. The dynamic networks, on the other hand, are used for less predictable or bursty forms of data movement, such as cache misses, some forms of data I/O, and operations that happen only occasionally.

Also unlike a more traditional multiprocessor on a chip where hardware handles many issues, the Raw device requires that much of its internal operation be handled by the compiler. For instance, cache coherency, cache misses, and the routing of data over the internal wiring must be handled by the compiler (or user).

2.2.2 PipeRench

Another example of a novel CGRA is PipeRench [174, 356], a project from Carnegie Mellon University. The chief goal of PipeRench was to develop an architecture that effectively employs run-time reconfiguration for hardware virtualization. Thus, an application that does not physically fit on a particular PipeRench's available resources can still execute. Using a coarse-grained architecture in this case greatly reduces the amount of configuration data that must be quickly swapped in and out of hardware regions.

Figure 2.15 illustrates the general architecture of this CGRA. The hardware is organized in pipeline stages called "stripes". Each stripe consists of 16 processing elements (or PEs) that contain 8-bit-wide logic and an 8-entry register file. The PEs within a stripe are all interconnected, reducing placement and routing issues. Note that the physical stripes are cleverly interleaved with the appropriate register file interconnections to create a ring structure out of the stripes—something that is important for virtualization. The on-chip logic for controlling the configuration and the virtualization functions is not shown in the figure.

With regards to PE logic, each PE contains shifters and multiplexers that can be configured to operate on inputs and the PE's main functional unit

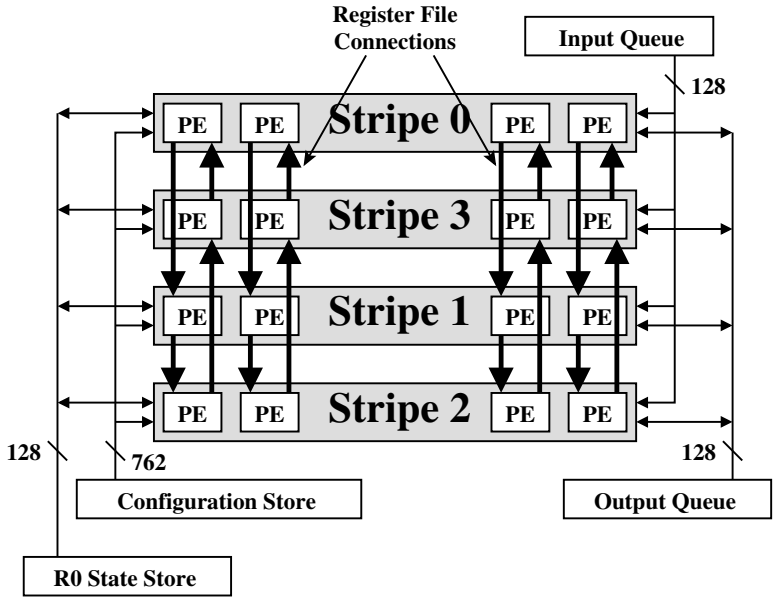

Fig. 2.15. PipeRench Architecture

is a collection of 8 3-bit LUTs, one for each bit of the operand width and each configured with the same function. In addition to the LUTs, specialized carry logic is also included to support fast addition. Due to the interconnect available, the PEs can be easily combined to form wider operations, including wide shifts using the input shifters. Only 42 bits are required to configure a PE while 672 bits are needed for an entire stripe.

As mentioned above, the register file provides 8 entries, but it also is a key part of the communication of data within the architecture. A PE's output can be written to any of the 8 registers and those registers not written to by the PE are overwritten by the previous stripe's values for those registers—basically, implementing the pipelining of operands. Further, Register 0 (R0) plays a special role. If a stripe operates as the first stage of a virtual pipeline, its R0 registers are loaded by the global input bus to provide the data for the pipeline. If a stripe operates as the last stage of a virtual pipeline, then the R0 registers' outputs are written to the global output bus.

Regarding virtualization, if a virtual stripe must be swapped out of a physical stripe to make room for more logic, the R0 registers for the stripe are stored away in the R0 State Store before reconfiguring the stripe. Likewise, when a virtual stripe is restored into a physical stripe, its configuration and R0 state are restored from the Configuration and R0 State Stores, respectively. One pass through the virtual pipeline is made for each set of new inputs to the first virtual pipeline stripe. With its interconnection architecture and this state configuration/restoration support, the PipeRench architecture is thus

constructed to easily support PipeRench applications across a wide range of PipeRench devices having different numbers of physical stripes. The actual device fabricated by CMU was a 16 physical stripe device with the ability to store 256 virtual stripes [356].

Considering the communication channels mentioned above, PipeRench supports applications with only limited feedback. Despite this, many datapath-oriented applications—such as FFTs, DCTs, and many encryption algorithms—require only feed-forward structures and map reasonably well to the architecture.

2.2.3 RaPiD

Another important linear CGRA architecture called RaPiD [99–101, 134] (for Reconfigurable Pipelined Datapath) was developed by the University of Washington. The goal of the architecture were to develop a high-performance coarse-grained reconfigurable device that could be targeted to specific application domains and could support application development at a relatively high level (i.e., not doing hardware design as with FPGAs). To help with the second of these two goals, RaPiD's architecture and application development system were co-developed to ease application development for the end user. The RaPiD architecture and related domain-specific reconfigurable architectures continues as a topic of research even today [86, 327].

As illustrated in Figure 2.16, this architecture is structured so that data is streamed through a mix of different coarse-grained function units having a common data width (generally between 8- to 32-bits wide) and the interconnection between these function units is through a single, flexible routing channel. A "Streams Manager" provides the architecture with streams of data from external memory or other sources and likewise receives the output streams from the CGRA and writes the data to external devices. During operation, the data pipeline does not need to be statically configured—the RaPiD architecture allows some resources' configurations (e.g., input muxes, ALU functions, etc.) to be altered during the operation of the application so the data flow can be somewhat dynamic, changing based on the application's needs. The configuration of the dynamically changeable resources is controlled cycle by cycle using the "Instruction Generator" and "Configurable Instruction Decoder."

As for the function units, the mix and number of function units is chosen based on the application domain for which the device will be used. The mix of function units is not necessarily limited to ALUs, multipliers, registers, and RAM, but can be any function that can be well utilized in a given application domain. For instance, a RaPiD for the communications application domain might have a Viterbi decoder as a function unit. A configurable delay element providing a zero to three-stage delay exists at the output of each unit to help with the scheduling of operations within the pipeline.

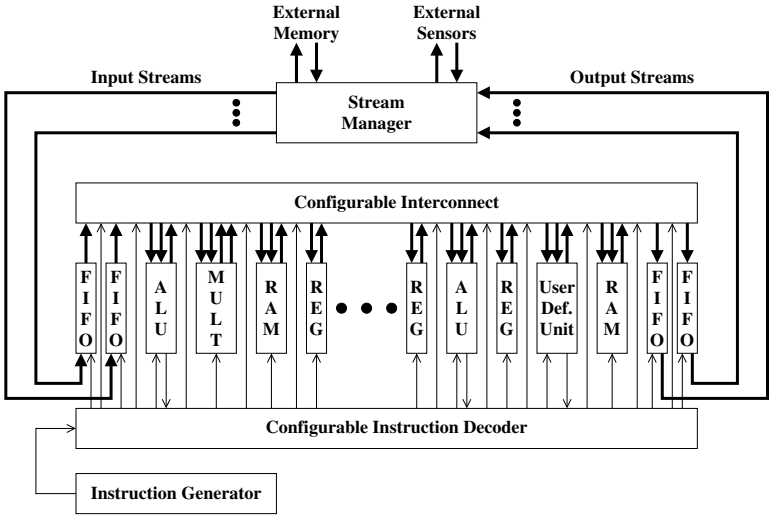

Fig. 2.16. RaPiD Architecture

The routing channel has segments of various lengths to support communications at different distances between function units. Further, as mentioned above, the interconnection among modules can be dynamic such as through selecting different inputs using the function units' input multiplexers. Unlike the other coarse-grained architectures we have described, some of the routing can be driven in one of two directions, allowing for feedback and flexibility in the mapping of applications. The direction a routing segment is driven, though, cannot be dynamically changed.

The "Stream Manager", which produces the input data and consumes the output data, is essentially a memory interface with an address generator and FIFO for each input or output stream (the FIFOs are explicitly illustrated in Figure 2.16). The flow of data through the architecture and instruction generation (or sequencing) are decoupled. A RaPiD array does provide a synchronization mechanism, though: the RaPiD array is halted upon the read of an empty input FIFO or the write of a full output FIFO.

2.2.4 PACT XPP

As an example of a commercial CGRA, the eXtreme Processing Platform (XPP) [36,317] is a computing array with a data-driven processing model and hierarchical configuration management developed by PACT Informationstechnologie GmbH. Like many CGRA architectures, the PACT XPP was developed to handle streaming data applications such as signal or media processing. As a product, PACT provides the CGRA as intellectual property to their customers for developing custom VLSI designs—a common model for commercial CGRA products.

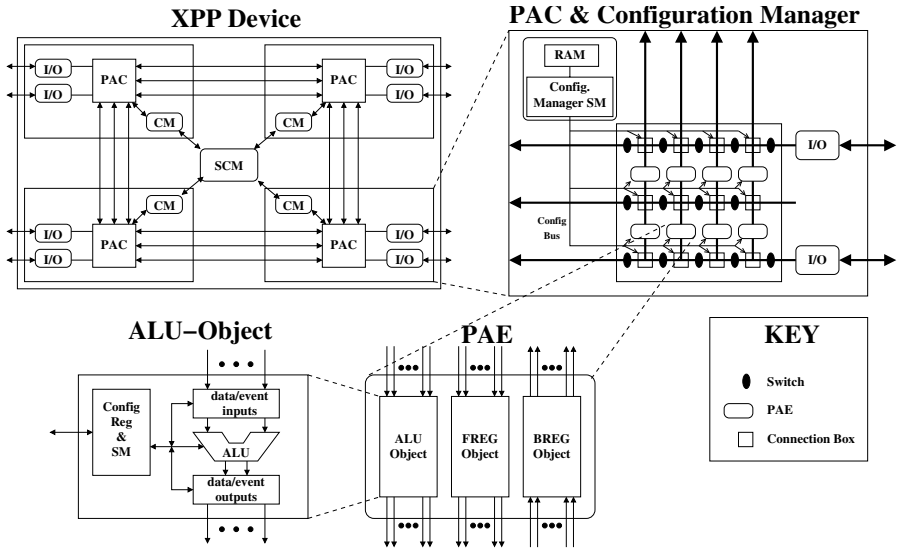

Fig. 2.17. PACT's eXtreme Processing Platform

Figure 2.17 illustrates the architecture at four different levels. The top level is the XPP device, which consists of several Processing Array Clusters (PACs) with their associated configuration management hardware. The configuration management (CM) hardware includes a state-machine-based controller and local RAM. At the device level, a supervising CM (or SCM) controls the overall configuration of the device and this SCM can yet be controlled by SCMs external to the device, creating a configuration management hierarchy.

At the next-level down, the PAC itself is an array of Processing Array Elements (PAEs), connection boxes for routing between vertical and horizontal busses, and switches for segmenting the horizontal busses. Each PAC also includes I/O resources as well.

At the third level, each PAE contains three objects: a function object, a forward register (FREG), and a backward register (BREG). In Figure 2.17, an ALU Object is the function object, but other objects, such as RAMs are possible. The FREG and BREG objects provide vertical routing support as well as data control flow (FREG), counters (FREG), adders/subtracters (BREG), and barrel shifters (BREG).

At the lowest level, the ALU object can consume and produce data and event packets that are key to the data-driven computation model used by the architecture. Data packets, of course, contain the results of an operation while the event packets contain the condition or state bits resulting from the operation. The width of the data packets are the width of the basic architecture (e.g., 24 or 32 bits) while the width of event packets is only a few bits. Note that the configuration portion of the architecture can be affected

by ALU operations and the event packets, allowing for event-driven initiation of configuration or reconfiguration.

The data-driven computation model is used in the XPP to ensure ease of application development—instead of having to worry about the exact timing of operations, the operators only process when all of their input packets are available and their last output packet has been consumed by the next PAE. This data-driven model has also been augmented with additional handshaking between PAEs and CMs so that a CM knows when a PAE can be reconfigured and when it is busy with a computation.

The hierarchical nature of configuration management has several effects. First, it provides a way for scaling the configuration of large systems or devices. Next, its allows configuration to be performed independently in the different chip regions. A corresponding result is that different portions of the XPP can be executing unrelated functions. Finally, the self-configuration capability can also be used either locally or globally throughout the system.

2.2.5 MathStar

Another commercial example of a CGRA architecture is the Field-Programmable Object Array (FPOA) produced by MathStar [203]. Like RaPiD, the MathStar architecture is intended to be optimized for a particular application domain and customers can specify the needed mixes of function units (called Silicon Objects) to meet their application needs. To make this a cost effective and fast time-to-market proposition, MathStar's FPOA is structured so that any function unit type can fit at any position in the device's two-dimensional object array. The result of this engineering is that an FPOA with a custom function unit mix can be ready for fabrication in less than a month with 1-GHz internal operation speeds without having to solve any additional analog signaling issues since the Silicon Objects are pre-engineered to deal with such issues.

Figure 2.18 is a conceptual illustration of the architecture. As mentioned before, the array can be heterogeneous or homogeneous, depending on the particular mix of Silicon Objects chosen for the array. The array given in the figure is a mix of register files, ALUs, and multiply/accumulate units. Note that the architecture also supports various I/O standards (including high-speed serial I/O) as well as internal RAM. The available Silicon Objects also include a logic block consisting of four four-input LUTs, a CRC generator, content addressable memories, and external memory interfaces.

As for the routing architecture, the FPOA supports 21-bit busses to communicate 16 data bits, a one-bit data-valid flag, and 4 control/state bits. Generally, the data and data-valid bits are handled as a unit while the control bits can be configured independently. Depending on the Silicon Object, the control/state bits can provide such information as the sign of the data, a carry bit, or a start-of-packet marker or they can be used to control the function of

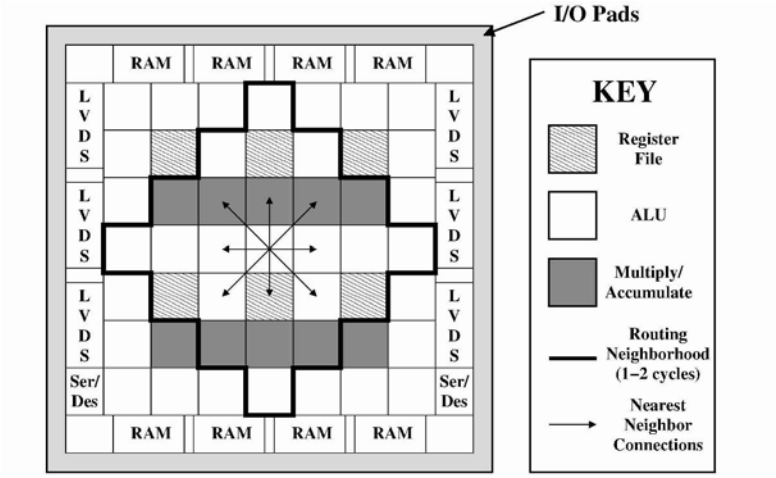

Fig. 2.18. MathStar's FPOA Architecture

the Silicon Objects, such as selecting the function of the ALU or the modes for the multiply-accumulate unit.

As the arrows in Figure 2.18 illustrate, each Silicon Object can communicate directly with its 8 immediate neighbors with, at most, four unique values. The values are available within a single clock cycle. The figure also illustrates that, with one level of pipelining, a Silicon Object can route its signal to any of 24 other cells within its extended neighborhood. Using more levels of pipelining, a Silicon Object's output signals can reach the rest of the array.

2.3 Summary

In this chapter, we have discussed the internal structure of FPGAs. Starting from simple homogeneous tiled arrays of logic blocks, I/O blocks, and interconnect, FPGAs have become complex systems on a chip, with elaborate logic clusters, a rich memory hierarchy, dedicated arithmetic function units, and high-speed serial I/O. The capacity of FPGAs has also greatly increased, which makes possible larger reconfigurable computing applications. The ability to reprogram SRAM-based FPGAs, either entirely or partially, is also an important feature for reconfigurable computing. Finally, we discussed how new coarse grained architectures have been developed that trade off FPGA flexibility for increased performance and lower power within specific application domains.

http://www.springer.com/978-0-387-26105-8

Reconfigurable Computing
Accelerating Computation with Field-Programmable Gate Arrays
Gokhale, M.B.; Graham, P.S.
2005, X, 238 p., Hardcover
ISBN: 978-0-387-26105-8