Field-Programmable Gate Array Architecture

Andrew Boutros and Vaughn Betz

Contents

Abstract

Since their inception more than thirty years ago, field-programmable gate arrays (FPGAs) have grown more complex, more capable, and more diverse in their applications. FPGAs can be reprogrammed at a fundamental level, changing the function and interconnection of millions of elements. By reconfiguring their hardware to match the application, FPGAs often achieve higher energy efficiency, lower latency or faster time-to-market across a very wide range of application domains. A modern FPGA combines many components, from logic

A. Boutros · V. Betz (✉)
Department of Electrical and Computer Engineering (ECE), University of Toronto, Toronto, ON, Canada
e-mail: andrew.boutros@mail.utoronto.ca; vaughn@eecg.utoronto.ca; vaughn@ece.utoronto.ca

A. Chattopadhyay (ed.), *Handbook of Computer Architecture*,
https://doi.org/10.1007/978-981-15-6401-7_49-1

blocks, programmable routing and memory blocks to networks-on-chip and processor subsystems. For best efficiency, each component must be carefully architected to match the needs of a wide range of applications, and to mesh well with the other components. Their design involves many different choices starting from the high-level architectural parameters down to the transistor-level implementation details. This chapter describes the evolution of these FPGA components, their design principles and implementation challenges.

Keywords

FPGA architecture · Reconfigurable computing · Programmable logic · FPGA applications

Introduction

The idea of reconfigurable computing originated in the 1960s with the Fixed Plus Variable Structure Computer (Estrin 1960), which aimed to enhance a conventional computing system with the capability to *temporarily* morph into a more application-specialized architecture. This would be achieved using additional programmable logic and interconnect circuitry to implement operations beyond the capabilities of the fixed datapath processor. A variety of research efforts subsequently investigated ideas for reconfigurable computer architectures that could combine both software-like flexibility and hardware efficiency. However, it was over 20 years later that the first commercially successful reconfigurable computing device, known as a field-programmable gate array (FPGA), was created by Xilinx in 1985.

As illustrated in Fig. 1, FPGAs consist of a two-dimensional array of programmable blocks (logic, IO, and others) that can be flexibly connected using a network of pre-fabricated wires with programmable switches between them. The functionality of all the FPGA blocks and the connectivity of routing switches are controlled using millions of configuration bits, usually stored in static random access memory (SRAM) cells, that can be configured to implement arbitrary digital circuits. The designer describes the desired functionality in a hardware description language (HDL) such as Verilog/VHDL or possibly uses high-level synthesis to translate a higher-level programming language (e.g., C++ or OpenCL) to HDL. The HDL design is then compiled using a complex computer-aided design (CAD) flow into the *bitstream* file, analogous to a software program executable, which is used to program all the FPGA's configuration SRAM cells.

FPGAs combine aspects of general-purpose processors and application-specific integrated circuits (ASICs). Their programmability allows a single FPGA to implement many different applications similar to a software-programmable processor, while the fact that their *hardware* is reconfigurable enables custom systems similar to an ASIC. However, FPGAs have a significantly lower non-recurring engineering cost and shorter time-to-market since they do not require the physical design, layout, fabrication, and verification stages that a custom ASIC would normally go through.

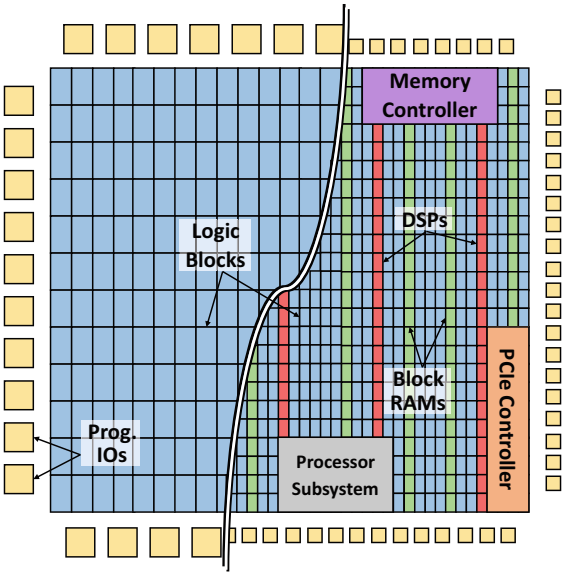

Fig. 1 Early FPGA architecture with programmable logic and IOs vs. modern heterogeneous FPGA architecture with RAMs, DSPs, and other hard blocks. All blocks are interconnected using bit-level programmable routing

A pre-fabricated off-the-shelf FPGA can be used to implement a complete system in a matter of weeks, and also enables continuous hardware upgrades to support new features or fix bugs by simply loading a new bitstream after deployment in-field, thus the name *field-programmable*. This makes FPGAs a compelling solution for medium and small volume designs, especially with the fast-paced product cycles in today's markets. FPGAs can also implement the exact hardware needed for each application (e.g., datapath bitwidth, pipeline stages, number of parallel compute units, memory subsytem, etc.) instead of the fixed one-size-fits-all architecture of general-purpose processors (CPUs) or graphics processing units (GPUs). Consequently, they can achieve higher efficiency than CPUs or GPUs by implementing instruction-free streaming hardware (Hall and Betz 2020) or a processor *overlay* with an application-customized pipeline and instruction set (Boutros et al. 2020).

However, the flexibility of FPGA hardware comes with an efficiency cost compared to ASICs. Kuon and Rose (2007) show that circuits using only the FPGA's programmable logic blocks average 35× larger and 4× slower than corresponding ASIC implementations. A more recent study (Boutros et al. 2018) shows that for full-featured designs which heavily utilize the other FPGA blocks (e.g., RAMs and DSPs), this area gap is reduced to 9×. FPGA architects seek to reduce this efficiency gap as much as possible while maintaining the programmability that makes FPGAs useful across a wide range of applications.

This chapter introduces key principles of FPGA architecture and highlights the evolution of these devices over the past three decades. Figure 1 shows how FPGAs evolved from simple arrays of programmable logic and IO blocks to complex heterogeneous multi-die systems with embedded block RAMs (BRAMs), digital

signal processing (DSP) blocks, processor subsystems, diverse high-performance external interfaces, system-level interconnect, and more. The chapter first gives a brief overview of the CAD flows and methodology used to evaluate new FPGA architecture ideas, as well as key applications for FPGAs. Next, the design principles and challenges for each of the key components of an FPGA architecture are detailed, along with major innovations and future challenges.

Methodology and Tools for FPGA Architecture Evaluation

Figure 2 shows a simplified view of the FPGA architecture evaluation flow. It consists of three main components: a set of benchmark applications, an architecture model, and a CAD system. Unlike an ASIC built for a specific functionality, an FPGA is a general-purpose device that is designed for many use cases, some of which may not even exist when the FPGA is architected. Therefore, a candidate FPGA architecture is evaluated based on its efficiency when used to implement a variety of benchmark designs that are representative of the key FPGA markets and application domains. Typically, each FPGA vendor has a carefully selected set of benchmark designs collected from proprietary system implementations and different customer applications. There are also several open-source benchmark suites such as the classic MCNC20, the VTR (Murray et al. 2020a), and the Titan23 (Murray et al. 2013) suites which are commonly used in academic FPGA architecture and CAD research. While early academic FPGA research used the MCNC suite of designs, these circuits are now too small (thousands of logic primitives) and simple (only IOs and logic) to represent modern FPGA applications. The VTR and particularly the Titan suites are larger and more complex, making them more representative. As FPGA capacity grows and key applications change, new benchmark suites will continue to be needed to drive both FPGA architecture and CAD research.

The second component of the evaluation flow is the FPGA architecture model. The design of an FPGA involves many different decisions from architecture-

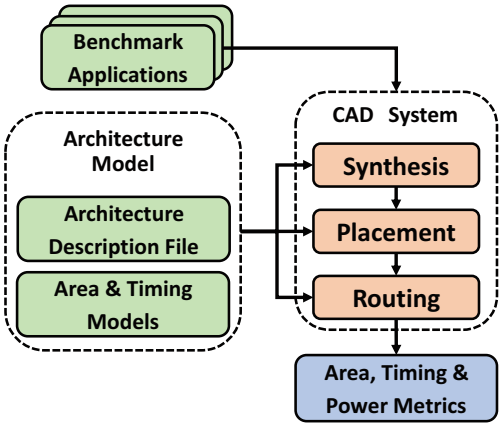

Fig. 2 FPGA architecture evaluation flow

level organization (e.g., number and type of blocks, distribution of wire segment lengths, size of logic clusters and logic elements), to micro-architectural details (e.g., DSP and BRAM modes of operation, hard arithmetic in logic blocks, switch block patterns), and down to transistor-level circuit implementation (e.g., programmable switch type, routing buffer transistor sizing, register implementation). It also involves different implementation styles; the logic blocks and programmable routing are designed and laid out as full-custom circuits, while most hardened blocks (e.g., DSPs) mix standard-cell and full-custom design for the block core and peripherals, respectively. Some blocks, such as BRAMs and high-speed IO, even include significant analog circuitry. All these different components need to be carefully modeled to evaluate the FPGA architecture in its entirety. Tools such as COFFE (Yazdanshenas and Betz 2019) were developed to automate the transistor-level design and modeling of FPGA blocks and programmable routing components, speeding up FPGA architecture investigations. The area, timing, and power models for each of these components are then typically combined in an architecture description file, along with a specification of how these blocks and routing components are organized into the overall architecture.

Finally, a re-targetable CAD system such as the Verilog-to-Routing (VTR) flow (Murray et al. 2020a) is used to map the selected benchmarks to the described FPGA architecture. Such a CAD system consists of a sequence of complex optimization algorithms that synthesizes a benchmark written in an HDL into a circuit netlist, maps it to the different FPGA blocks, places the mapped blocks at specific locations on the FPGA, and routes the connections between them using the specified programmable routing architecture. The implementation produced by the CAD system is then used to evaluate several key metrics. Total area is the sum of the areas of the FPGA blocks used by the application, along with the programmable routing included with them. A timing analyzer finds the critical path(s) through the blocks and routing to determine the maximum frequencies of the application's clocks. Power consumption is estimated based on resources used and signal toggle rates. FPGAs are never designed for only one application, so these metrics are averaged across all the benchmarks. Finally, the overall evaluation blends these average area, delay, and power metrics appropriately depending on the architecture goal (e.g., high performance or low power). Other metrics such as CAD tool runtime and routability of the benchmarks on a candidate architecture are also often considered.

As an example, a key set of questions in FPGA architecture is: What functionality should be *hardened* (i.e., implemented as a new ASIC-style block) in the FPGA architecture? How flexible should this block be? How much of the FPGA die area should be dedicated to it? Ideally, an FPGA architect would like the hardened functionality to be usable by as many applications as possible at the least possible silicon cost. An application that can make use of the hard block will benefit by being smaller, faster and more power-efficient than a *soft* implementation that uses only the programmable logic and routing. This motivates having more programmability in the hard block to capture more use cases; however, higher flexibility generally comes at the cost of larger area and reduced efficiency of the hard block. On the other hand, if a hard block is not usable by an application circuit, its silicon area is

wasted; the FPGA user would rather have more of the usable general-purpose logic blocks in the area of the unused hard block. The impact of this new hard block on the programmable routing must also be considered – does it need more interconnect or lead to slow routing paths to and from the block? To evaluate whether a specific functionality should be hardened or not, both the cost and gain of hardening it have to be quantified empirically using the flow described in this section. FPGA architects may try many ideas before landing on the right combination of design choices that adds just the right amount of programmability to make this new hard block a net win.

In the rest of this chapter, we detail many different components of FPGAs and key architecture questions for each. While we describe the key results without detailing the experimental methodology used to find them, in general they came from a holistic architecture evaluation flow similar to that in Fig. 2.

Key FPGA Applications

In this section, we present some of the key application domains of FPGAs and highlight their advantages for use cases in these areas.

Wireless communications and (e.g., cell phone base stations) is a very large market for FPGAs. The **reconfigurability** of FPGAs allows service providers to implement a range of different standards and upgrade them in-field, while achieving much higher **energy efficiency** compared to instruction-set-based DSP devices. One of the key components in wireless communications is signal processing, such as filtering. The **direct hardware execution** (without an instruction stream) of FPGAs makes them very efficient for repetitive tasks of this nature. Table 1 compares the performance, power and energy efficiency of a Stratix IV FPGA to two Texas Instruments DSP devices (scaled optimistically to the same 40 nm process technology of the FPGA) when implementing simple signal filtering using a 51-tap finite impulse response (FIR) filter. The results show that even a single instance of the FIR filter (consuming only 2% of the FPGA resources) can achieve two orders of magnitude higher performance compared to both DSPs, and 7.7× and 63.2× higher energy efficiency compared to the C5505 and C674x, respectively. The FPGA can achieve another order of magnitude higher performance by instantiating up to 49 instances of the FIR filter working in parallel at the cost of only 9× higher power consumption since the FPGA static power (80% of the FPGA power in Table 1) remains the same regardless of the amount of utilized resources.

Table 1 Performance, power, and energy efficiency comparison between a Stratix IV FPGA and two DSP devices. The results of the DSPs are optimistically scaled to the FPGA's 40 nm process technology

Device	Performance (Samples/s)	Power (mW)	Energy efficiency (Samples/W)
TI C5505	1.77×10^6	28	6.32×10^7
TI C674x	3.21×10^6	416	7.72×10^6
Stratix IV GX230	5.1×10^8	1046	4.88×10^8

Wired communications and networking are also heavy users of FPGAs. The **richness and diversity of FPGA IOs** are important in this use case, as many different and very high-speed IO standards are used in chip-to-chip, server-to-server and city-to-city communications. FPGAs are often used in high-end packet processing and switching systems, which have a high degree of parallelism and a need for high throughput and low latency (Zhao et al. 2020). This is well-suited to an FPGA's **spatial architecture** and the **ability to customize processing pipelines** to the bare minimum required by the target application to reduce latency compared to general-purpose processors with a fixed pipeline and memory hierarchy. The **hardened network transceivers** in modern FPGAs along with the ability to customize the network stack implementation also make FPGAs suitable for ultra-low latency networking interfaces. This can also be useful in other domains, including financial applications such as high-frequency trading (Lockwood et al. 2012) where FPGA reprogrammability allows integration of the rapidly changing trading algorithms on the same chip as the low-latency networking interface.

More recently, FPGAs have also been deployed on a large scale in datacenters where both their computation and networking capabilities are leveraged. The Microsoft Catapult project couples every CPU server with an FPGA that can be used to accelerate search engines, packet processing, encryption and compression (Putnam et al. 2014; Caulfield et al. 2016). This achieved a 95% improvement in the ranking throughput of their search engine infrastructure at the cost of only 10% higher power consumption. The network-connected FPGAs in the Catapult project were also used to implement Brainwave, a datacenter-scale deep learning accelerator for real-time low-latency inference (Fowers et al. 2018).

The **hardware reprogrmmability** of FPGAs has led to their extensive use in ASIC prototyping (Krupnova and Saucier 2000), where either a part or the entirety of an ASIC design is emulated on FPGAs to verify functionality or estimate performance before fabrication. There are a myriad of other application domains for FPGAs including embedded real-time video processing in autonomous vehicles (Rettkowski et al. 2017), genomics (Turakhia et al. 2018), biophotonic simulations (Young-Schultz et al. 2020), accelerated RTL simulation (Karandikar et al. 2018), and many more.

These diverse applications are enabled by the various components of an FPGA architecture working together, and in the following sections we detail the architecture of each of these components.

Programmable Logic Blocks

A fundamental component of an FPGA is the programmable logic block that can implement arbitrary logic functions.

The earliest reconfigurable computing devices were *programmable array logic* (PAL) architectures. As shown in Fig. 3, PALs consisted of an array of `AND` gates feeding another array of `OR` gates which could implement any Boolean logic expression as a two-level sum-of-products function. Programmable switches are

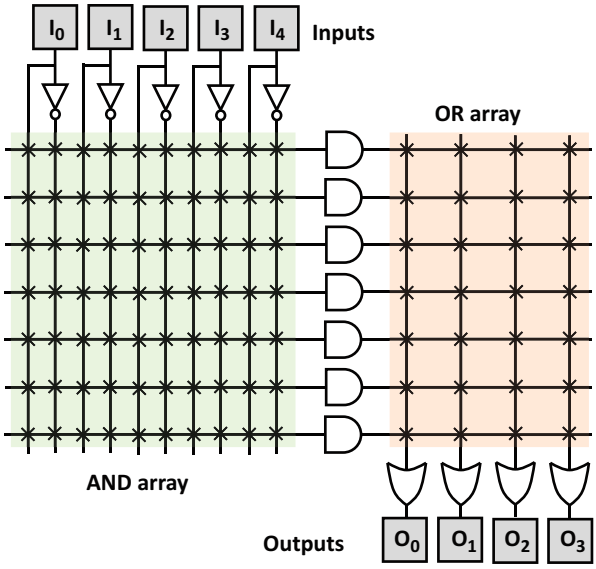

Fig. 3 Programmable array logic (PAL) architecture with an AND array followed by an OR array. The crosses are reconfigurable switches that are programmed to implement any Boolean expression as a two-level sum-of-products function

flexibly configured to select the inputs to each of the AND/OR gates to implement different Boolean expressions. The design tools for PALs were very simple since the delay through the device is constant regardless of the logic function implemented. However, PALs did not scale well; as device logic capacity increased, the wires connecting the AND/OR grid became increasingly longer and slower and the number of required programmable switches grew quadratically.

Subsequently, *complex programmable logic devices* (CPLDs) kept the AND/OR arrays as the basic logic elements, but attempted to solve the scalability challenge by integrating multiple PALs on the same die with a crossbar interconnect between them at the cost of more complicated design tools. Shortly after, Xilinx pioneered the first FPGA in 1985, which consisted of an array of SRAM-based lookup tables (LUTs) with programmable interconnect between them. This style of reconfigurable devices was shown to scale very well, with LUTs achieving much higher area efficiency compared to the AND/OR logic in PALs and CPLDs. Consequently, LUT-based architectures became increasingly dominant and today LUTs form the fundamental logic element in all commercial FPGAs. Some research attempts investigated replacing LUTs with a different form of configurable AND gates: a full binary tree of AND gates with programmable output/input inversion known as an *AND-inverter cone* (AIC) (Parandeh-Afshar et al. 2012). However, when thoroughly evaluated in Zgheib et al. (2014), AIC-based FPGA architectures had significantly larger area than LUT-based ones, with delay gains only on small benchmarks that have relatively short and localized critical paths.

A K-LUT can implement any K-input Boolean function by storing its truth table in 2^K configuration SRAM cells. Figure 4a shows the transistor-level circuit implementation of a 4-LUT using pass-transistor logic. The four inputs (A, B, C, and D) are used as multiplexer select lines to choose an output from the 16 values

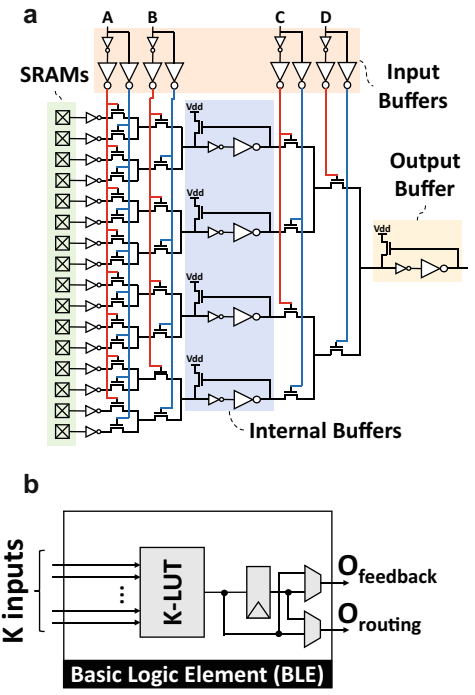

Fig. 4 (**a**) Transistor-level implementation of a 4-LUT with internal buffers between the second and third LUT stages, and (**b**) Basic logic element (BLE)

of the truth table in the SRAMs. In addition to the output buffer, an internal buffering stage (shown between the second and third stages of the LUT in Fig. 4a) is typically implemented to mitigate the quadratic increase in delay when passing through a chain of pass-transistors. The sizing of the LUT's pass-transistors and the internal/output buffers is carefully tuned to achieve the best area-delay product. Classic FPGA literature (Betz et al. 1999) defines the *basic logic element* (BLE) as a K-LUT coupled with an output register and 2:1 bypassing multiplexers as shown in Fig. 4b. Thus, a BLE can be used to implement just a flip-flop (FF) with the LUT configured as an identity function, or any Boolean expression with up to K inputs and optionally-registered output. As illustrated in Fig. 5a, BLEs are typically clustered in *logic blocks* (LBs), such that an LB contains N BLEs along with local interconnect. The local interconnect in the logic block consists of multiplexers between signal sources (BLE outputs and logic block inputs) and destinations (BLE inputs). These multiplexers are often arranged to form a local full (Betz and Rose 1998) or partial (Lemieux et al. 2000) crossbar. At the circuit level, these multiplexers are usually built as two levels of pass transistors, followed by a two-stage buffer as shown in Fig. 5b; this is the most efficient circuit design for FPGA multiplexers in most cases (Chiasson and Betz 2013a). Figure 5a also shows the switch and connection block multiplexers forming the programmable routing used for inter-LB communication; this routing is discussed in detail in the "Programmable Routing" section.

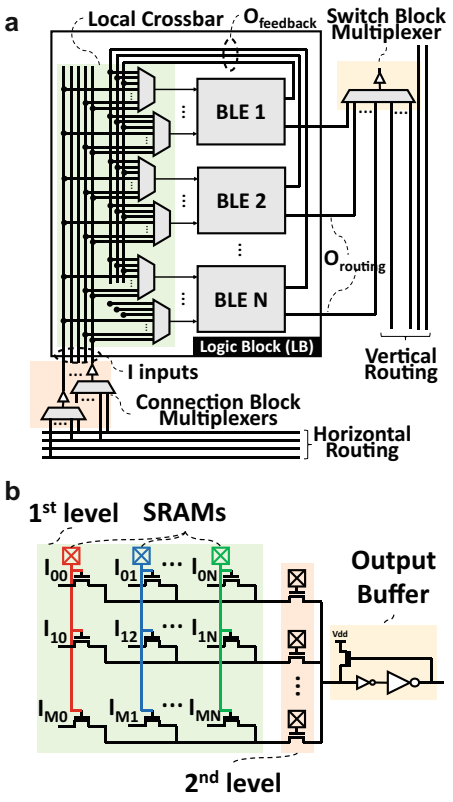

Fig. 5 (**a**) Logic block (LB) internal architecture, and (**b**) Two-level multiplexer circuitry

Over the years, the size of LUTs (K) and LBs (N) have gradually increased with growing device logic capacity. As K increases, more functionality can be captured into a single LUT. Therefore, the same circuit can be implemented using fewer LUTs with a smaller number of logic levels on the critical path, which increases performance. In addition, the demand for inter-LB routing decreases as more connections are captured into the fast local interconnect by increasing N. On the other hand, the area of the LUT increases exponentially with K (due to the 2^K SRAM cells) and its speed degrades linearly (due to propagation through a chain of K pass transistors with periodic buffering). The size of the local crossbar also increases quadratically and its speed degrades linearly with increasing N. Ahmed and Rose (2004) empirically evaluated these trade-offs and found that LUTs of size 4–6 and LBs of size 3–10 BLEs offer the best area-delay product for an FPGA architecture, with 4-LUTs leading to a better area but 6-LUTs yielding a higher speed. Historically, the first Xilinx FPGAs had an LB with only two 3-LUTs (i.e., $N = 2$, $K = 3$). LB size gradually increased over time and by 1999, Xilinx's Virtex family had four 4-LUTs and Altera's Apex 20K family had ten 4-LUTs in each LB.

The next major logic feature was the *fracturable* LUTs introduced in 2003 by Altera in their Stratix II architecture. Ahmed and Rose in (2004) showed that

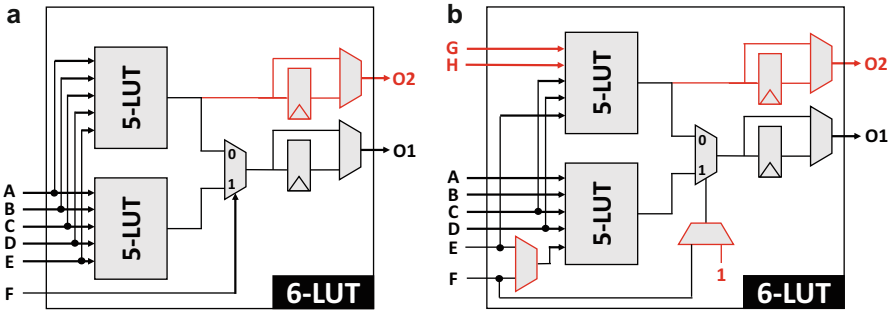

Fig. 6 6-LUT fracturable into two 5-LUTs with (**a**) no additional input ports, leading to 5 shared inputs or (**b**) two additional input ports and steering multiplexers, leading to only 2 shared inputs

an LB with ten 6-LUTs achieved 14% better performance but increased area by 17% compared to an LB with ten 4-LUTs. In addition, an architecture with only 6-LUTs can suffer from significant under-utilization. Lewis et al. found that 64% of the LUTs implemented for a commercial benchmark suite used fewer than 6 inputs, wasting some of the 6-LUT functionality (Lewis et al. 2005). Based on these observations, fracturable LUTs were introduced to combine the best of both worlds: the higher performance of larger LUTs and the superior area-efficiency of smaller ones. A fracturable $\{K, M\}$-LUT can be configured as a single LUT of size K or can be fractured into two LUTs of size up to $K-1$ that collectively use no more than $K + M$ distinct inputs. Figure 6a shows that a 6-LUT is internally composed of two 5-LUTs plus a 2:1 multiplexer. Consequently, almost no circuitry (only the red added output) is necessary to allow a 6-LUT to instead operate as two 5-LUTs that share the same inputs. However, this requires the two 5-LUTs to share all their inputs which limits how often both LUTs can be simultaneously used. Adding extra routing ports as shown in Fig. 6b relaxes this constraint and makes it easier to find two logic functions that can be packed together into a fracturable 6-LUT at the cost of slightly increasing its area. The *adaptive logic module* (ALM) in the Stratix II architecture implemented a {6, 2}-LUT that had 8 input and 2 output ports. Thus, an ALM can implement a 6-LUT or two 5-LUTs sharing 2 inputs (and therefore a total of 8 distinct inputs). Pairs of smaller LUTs could also be implemented without any shared inputs, such as two 4-LUTs or one 5-LUT and one 3-LUT. With a fracturable 6-LUT, larger logic functions are implemented in 6-LUTs reducing the logic levels on the critical path and achieving better performance. On the other hand, pairs of smaller logic functions can be packed together (each using only half an ALM), improving area-efficiency. The LB in Stratix II not only increased the performance by 15%, but also reduced the logic and routing area by 2.6% compared to a baseline 4-LUT-based LB (Lewis et al. 2005).

Xilinx later adopted a similar approach in their Virtex-5 architecture in which the 6-LUTs can also be decomposed into two 5-LUTs. However, they adopted a LUT architecture similar to that shown in Fig. 6a with minimal changes compared to the traditional 6-LUT (i.e., no extra input routing ports or steering multiplexers).

This results in a lower area per fracturable LUT, but makes it more difficult to pack two smaller LUTs together as they must use no more than 5 distinct inputs. While subsequent architectures from both Altera/Intel and Xilinx have also been based on fracturable 6-LUTs, recent work from Microsemi (Feng et al. 2018) revisited the 4-LUT vs. 6-LUT efficiency trade-off for newer process technologies, CAD tools and designs than those used in Ahmed and Rose (2004). It shows that a LUT structure with two tightly coupled 4-LUTs, one feeding the other, can achieve performance close to conventional 6-LUTs while maintaining the high utilization and area efficiency of 4-LUTs. In terms of LB size, FPGA architectures from Altera/Intel and Xilinx converged on the use of relatively large LBs with ten and eight BLEs respectively, for several generations. However, the Versal architecture from Xilinx further increases the number of BLEs per LB to thirty two (Gaide et al. 2019). This significant increase in LB size is motivated by two main factors. First, inter-LB wire delay is scaling poorly with process shrinks, so capturing more connections within an LB's local routing is increasingly beneficial. Second, ever-larger FPGA designs tend to increase CAD tool runtime, but larger LBs can mitigate this trend by simplifying placement and inter-LB routing.

The number of FFs per BLE and the circuit-level FF implementation are other important architecture choices. Early FPGAs with non-fracturable LUTs had a single FF to optionally register the LUT output as shown in Fig. 4b. When they moved to fracturable LUTs, both Altera/Intel and Xilinx architectures added a second FF to each BLE so that both outputs of the fractured LUT could be registered, as shown in Fig. 6a and b. In the Stratix V architecture, the number of FFs was doubled (i.e., four FFs per BLE) to accommodate the increasing demand for FFs as designs became more deeply pipelined to achieve higher performance (Lewis et al. 2013). Low-cost multiplexing circuitry allows sharing the existing inputs between the LUTs and FFs to avoid adding more costly routing ports. Stratix V also implements FFs as pulse latches instead of edge-triggered FFs. As shown in Fig. 7b, this removes one of the two latches that would be present in a master-slave FF (Fig. 7a), reducing the register delay and area. A pulse latch acts as a cheaper FF with worse hold time as it latches the data input during a very short pulse instead of a clock edge as in conventional FFs. However, it would be area-inefficient to build a pulse generator for each latch. Instead, this cost is amortized

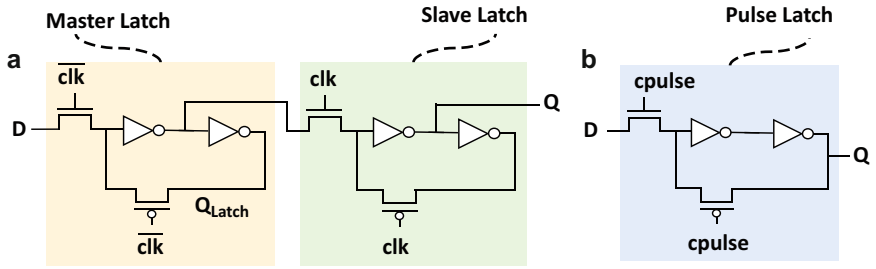

Fig. 7 Circuitry for (**a**) Master-slave positive-edge-triggered FF, and (**b**) Pulse latch

by implementing only two configurable pulse generators per LB; each of the 40 pulse latches in an LB selects which generator provides its pulse input. The FPGA CAD tools can also program the pulse width in these generators, allowing a limited amount of time borrowing between source and destination registers. Soon after, the Xilinx Ultrascale+ architecture also adopted the use of pulse latches as its FFs due to their area and speed benefits (Ganusov and Devlin 2016).

Murray et al. found that 22% of logic elements in the Titan suite of benchmarks implemented addition or subtraction (Murray et al. 2020b). When implemented with LUTs, each bit of arithmetic in a ripple carry adder requires two LUTs, one for generating the sum and another for the carry. This is inefficient as it results in high logic utilization and a slow critical path due to having many cascaded LUTs in series for computing the carries in multi-bit additions. Consequently, all modern FPGA architectures include hardened arithmetic circuitry in their LBs. There are many variants, but all have several common points. First, to avoid adding expensive routing ports, the arithmetic circuits re-use the LUT routing ports or are fed by the LUT outputs. Second, the carry bits are propagated on a special, dedicated interconnect with little or no programmability so that the crucial carry path is fast. The lowest cost arithmetic circuitry hardens ripple carry structures and achieves a large speed gain over LUTs (3.4× for a 32-bit adder in Murray et al. 2020b). Hardening more sophisticated structures like carry skip adders further improves speed (an additional 20% speed-up at 32 bits in Yazdanshenas and Betz 2019). The latest Versal architecture from Xilinx (Gaide et al. 2019) hardens the carry logic for 8-bit carry look-ahead adders (i.e., the addition can only start on every eighth BLE), while the sum, propagate and generate logic is all implemented in the fracturable 6-LUTs feeding the carry logic as shown in Fig. 8a. This organization allows implementing 1-bit of arithmetic per logic element. On the other hand, the latest Intel Agilex architecture can implement two bits of arithmetic per logic

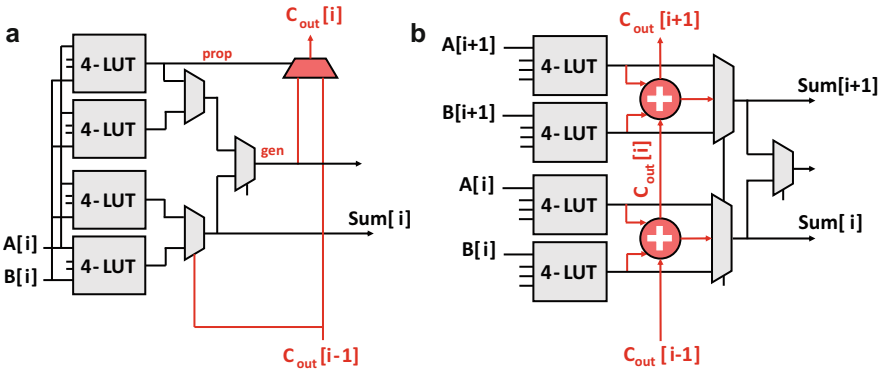

Fig. 8 Overview of the hard arithmetic circuitry (in red) in the logic elements of (**a**) Xilinx and (**b**) Altera/Intel FPGAs. $A[i]$ and $B[i]$ are the ith bits of the two addition operands A and B. The Xilinx logic elements compute carry propagate (`prop`) and generate (`gen`) in the LUTs, while the Altera/Intel ones use LUTs to pass inputs to the hard adders. Unlabeled inputs are unused when implementing adders

element, with a dedicated interconnect for the carry as shown in Fig. 8b. It achieves that by hardening 2-bit carry-skip adders that are fed by the four 4-LUTs contained within a fracturable 6-LUT (Chromczak et al. 2020). The study by Murray et al. (2020b) shows that the combination of fracturable LUTs and 2 bits of arithmetic (similar to that adopted in Altera/Intel FPGAs) is particularly efficient compared to architectures with non-fracturable LUTs or 1 bit of arithmetic per logic element. It also concludes that having dedicated arithmetic circuits (i.e., hardening adders and carry chains) inside the FPGA logic elements increases average performance by 75% and 15% for arithmetic microbenchmarks and general benchmark circuits, respectively.

Recently, deep learning (DL) has become a key workload in many end-user applications, with its core operation being multiply-accumulate (MAC). Generally, MACs can be implemented in DSP blocks as will be described in the "DSP Blocks" section; however, low-precision MACs with 8-bit or narrower operands (which are becoming increasingly popular in DL workloads) can also be implemented efficiently in the programmable logic (Caulfield et al. 2016). In this case, LUTs implement `AND` gates to generate partial products followed by an adder tree to reduce the partial products and perform the accumulation. Consequently, multiple recent studies (Rasoulinezhad et al. 2020; Eldafrawy et al. 2020) have investigated increasing the density of hardened adders in the FPGA's logic fabric to enhance its performance when implementing arithmetic-heavy applications such as DL. The work in Eldafrawy et al. (2020) proposed multiple different logic block architectures that incorporate 4 bits of arithmetic per logic element arranged in one or two carry chains with different configurations, instead of just 2 bits of arithmetic in an Intel Stratix-like ALM. These proposals do not require increasing the number of the (relatively expensive) routing ports in the logic clusters when implementing multiplications due to the high degree of input sharing in a multiplier array (i.e., for an N-bit multiplier, only $2N$ inputs are needed to generate N^2 partial products). The most promising of these proposals increases the density of MAC operations by $1.7\times$ while simultaneously improving their speed. It also reduces the required logic and routing area by 8% for general benchmarks, highlighting that more arithmetic density is beneficial for applications beyond DL.

Programmable Routing

Programmable routing commonly accounts for 50% or more of the fabric area and the critical path delay of applications (Chiasson and Betz 2013b), so its efficiency is crucial. Programmable routing is composed of pre-fabricated wire segments connected via programmable switches. By programming an appropriate sequence of switches to be on, a connection can be formed between any two function blocks. There are two main classes of FPGA routing architecture. *Hierarchical* FPGAs are inspired by the fact that designs are inherently hierarchical; higher-level modules instantiate lower-level modules and connect signals between them, with communication being more frequent between modules that are near each other in the design hierarchy. As shown in Fig. 9, hierarchical FPGAs can realize these

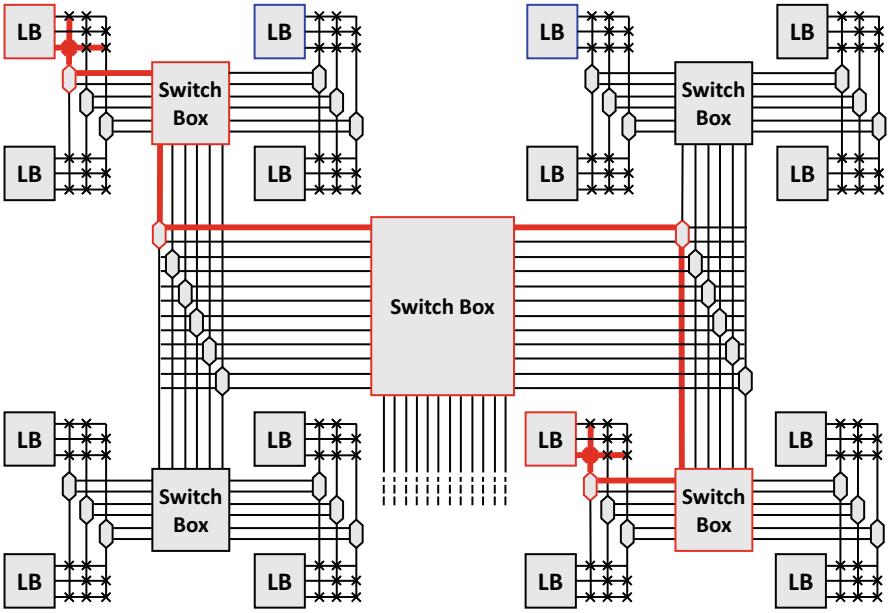

Fig. 9 Hierarchical routing architecture. A distant connection (highlighted in red) traverses through different levels of the hierarchy. Some blocks in physical proximity (highlighted in blue) still require several wires and switches to connect

connections with short wires that connect small regions of the chip. To communicate to more distant regions, a connection (highlighted in red) passes through multiple wires and switches as it traverses different levels of the interconnect hierarchy. This style of architecture was popular in many earlier FPGAs, such as Altera's Flex and Apex families, but it leads to very long wires at the upper levels of the interconnect hierarchy which became problematic as process scaling made such wires increasingly resistive. A strictly hierarchical routing architecture also results in some blocks that are physically close together (e.g., the blue blocks in Fig. 9) which still require several wires and switches to connect. Consequently, this routing architecture is primarily used today for smaller FPGAs, such as the FlexLogix FPGA IP cores that can be embedded in larger SoC designs.

The other type of FPGA interconnect is *island-style*, as depicted in Fig. 10. This architecture was pioneered by Xilinx and is inspired by the fact that a regular two-dimensional layout of horizontal and vertical directed wire segments can be efficiently laid out. As shown in Fig. 10, island-style routing includes three components: routing wire segments, connection blocks (multiplexers) that connect function block inputs to the routing wires, and switch blocks (programmable switches) that connect routing wires together to realize longer routes. The placement engine in FPGA CAD tools chooses which function block implements each element of a design in order to minimize the required wiring. Consequently, most connections between function blocks span a small distance and can be implemented with a few routing wires as illustrated by the red connection in Fig. 10.

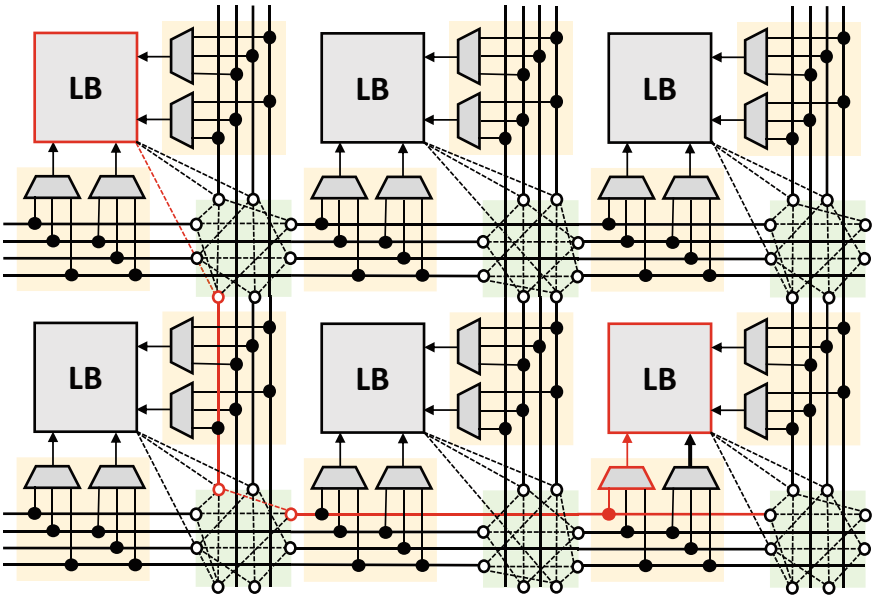

Fig. 10 Island-style routing architecture. Thick solid lines are routing wires while dashed lines are programmable switches. Connection and switch blocks are shaded in yellow and green, respectively

Creating a good routing architecture involves managing many complex trade-offs. It should contain enough programmable switching and wire segments that the vast majority of circuits can be implemented; however, too many wires and switches waste area and complicate the routing CAD problem. A routing architecture should also match the needs of applications. Ideally, short connections will be made with short wires to minimize capacitance and layout area, while long connections can use longer wiring segments to avoid the extra delay of passing through many routing switches. FPGA routing architecture design is also challenging as it involves many different and interacting choices. These choices include: how many routing wires each logic block input or output can connect to (F_c), how many other routing wires each wire can connect to (F_s), the lengths of the routing wire segments, the routing switch pattern, the electrical design of the wires and switches themselves, and the number of routing wires per channel (Betz et al. 1999). In Fig. 10 for example, $F_c = 3$, $F_s = 3$, the channel width is 4 wires, and some routing wires are of length 1, while others are of length 2. Fully evaluating these trade-offs and selecting the values for these architecture parameters for target applications and at a specific process node requires experimentation using a full CAD flow as previously discussed in the "Methodology and Tools for FPGA Architecture Evaluation" section.

Early island-style architectures incorporated only short wires that traversed a single logic block between programmable switches. Later research showed that this

resulted in more programmable switches than necessary, and that making all wiring segments span four logic blocks before terminating reduced application delay by 40% and routing area by 25% (Betz and Rose 1999). Modern architectures include multiple lengths of wiring segments to better match the needs of short and long connections, but the most plentiful wire segments remain of moderate length, with four logic blocks being a popular choice. Longer distance connections can achieve lower delay using longer wire segments, but in recent process nodes wires that span many (e.g., 16) logic blocks must use wide and thick metal traces on upper metal layers to achieve acceptable resistance (Petelin and Betz 2016). The amount of such long-distance wiring one can include in a metal stack is limited. To best leverage such scarce wiring, Intel's Stratix FPGAs allow long wire segments to be connected only to short wire segments, rather than function block inputs or outputs (Lewis et al. 2003). This creates a form of routing hierarchy within an island-style FPGA, where short connections use only the shorter wires, but longer connections pass through short wires to reach the long wire network. Another area where hierarchical FPGA concepts are used within island-style FPGAs is within the logic blocks. As illustrated in Fig. 5a, most logic blocks now group multiple BLEs together with local routing. This means that each logic block is a small cluster in a hierarchical FPGA; island-style routing interconnects the resulting thousands of logic clusters.

There has been a great deal of research into the optimal amount of switching, and how to best arrange the switches. While there are many detailed choices, a few principles have emerged. The first is that the connectivity between function block pins and wires (F_c) can be relatively low: typically only 10% or less of the wires that pass by a pin will have switches to connect to it. Similarly, the number of other wires that a routing wire can connect to at its end (F_s) can also be low, but it should be at least 3 so that a signal can turn left, right, or go straight at a wire endpoint. The local routing in a logic cluster (described in the "Programmable Logic Blocks" section) allows some block inputs and some block outputs to be swapped during routing (i.e., general programmable routing can deliver a signal to one of several input pins, which can then be routed to the right LUT input using the local crossbar). By leveraging this extra degree of flexibility and considering all the options presented by the multi-stage programmable routing network, the routing CAD tool can achieve high completion rates even with low F_c and F_s values. Switch patterns that give more options to the routing CAD tool also help routability; for example, the Wilton switch pattern ensures that following a different sequence of channels lets the router reach different wire segments near a destination block (Tang et al. 2019). Some recent architectures have also created L-shaped routing segments (formed by shorting a horizontal and vertical metal segment together) that allow connections between diagonally nearby blocks with fewer routing switches (Sivaswamy et al. 2005; Petersen et al. 2021).

There are also multiple options for the electrical design of programmable switches, as shown in Fig. 11. Early FPGAs used pass gate transistors controlled by SRAM cells to connect wires. While this is the smallest switch possible, the delay of routing wires connected in series by pass transistors grows quadratically,

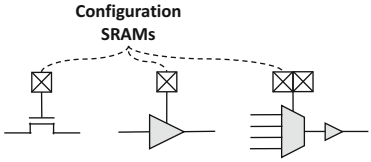

Fig. 11 Different implementations for SRAM-controlled programmable switches using pass transistors (left), tri-state buffers (middle), or buffered multiplexers (right)

making them very slow for large FPGAs. Adding some tri-state buffer switches costs area, but improves speed (Betz and Rose 1999). Most recent FPGAs primarily use a multiplexer built out of pass gates followed by a buffer that cannot be tri-stated, as shown in detail in Fig. 5b. The pass transistors in this *direct drive* switch can be small as they are lightly loaded, while the buffer can be larger to drive the significant capacitance of a routing wire segment. Such direct drive switches create a major constraint on the switch pattern: a wire can only be driven at one point, so only function block outputs and routing wires near that point can feed its routing multiplexer inputs and hence be possible signal sources. Despite this constraint, both academic and industrial work has concluded that direct drive switches improve both area and speed due to their superior electrical characteristics (Lewis et al. 2003; Lemieux et al. 2004). The exception is expensive or rare wires such as long wires implemented on wide metal traces on upper metal layers or the interposer-crossing wires discussed later in the "Interposers" section. These wires often have multiple tri-state buffers that can drive them, as the cost of these larger programmable switches is merited to allow more flexible usage of these expensive wires.

A major challenge for FPGA routing is that the delay of long wires is not improving with process scaling, which means that the delay to cross the chip is stagnating or increasing even as clock frequencies rise. This has led FPGA application developers to increase the amount of pipelining in their designs, thereby allowing multiple clock cycles for long routes. To make this strategy more effective, some FPGA manufacturers have integrated registers within the routing network itself. Intel's Stratix 10 device allows each routing driver (i.e., multiplexer followed by a buffer) to be configured as a pulse latch as shown in Fig. 7b, thereby acting as a register with low delay but relatively poor hold time. This allows deep pipelining of interconnect without using expensive logic resources, at the cost of a modest area and delay increase to the routing driver (Lewis et al. 2016). However, their poor hold time means using pulse latches in immediately consecutive Stratix 10 routing switches would lead to hold time violations, so not all of these interconnect registers can be simultaneously used. Therefore, Intel refined this approach in their Agilex devices by integrating actual registers (with better hold time) on only one-third of the interconnect drivers to mitigate the area cost (Chromczak et al. 2020). Rather than integrating registers throughout the interconnect, Xilinx's Versal devices instead add bypassable registers only on the inputs to function blocks. Unlike Intel's interconnect registers, these input registers are full-featured, with clock enable and clear signals (Gaide et al. 2019).

Since neighboring LB are likely to implement related logic, FPGA architectures also include dedicated interconnects to implement high-speed connections between adjacent LBs. Such connections are realized by allowing the outputs of an LB to drive the local input crossbar of its immediate neighbors without using the general programmable routing. The FPGA CAD tools can then decide how to place the implemented circuit such that critical inter-LB connections can benefit from these dedicated interconnects. To further increase the efficiency of the routing architecture, some recent studies build on this idea by analyzing a variety of benchmark circuits to extract the most commonly used routing patterns and implement them as dedicated routing structures (Nikolić et al. 2020). This results in a modest 3% improvement in the average critical path delay of the studied benchmarks, but these gains can be potentially improved through CAD enhancements to better exploit the dedicated routing.

Programmable IO

One of the unique properties of FPGAs is their programmable IO structures that allow them to communicate with a wide variety of other devices, making them the communications hub of many systems. For a single set of physical IOs to *programmably* support many different IO interfaces and standards, it requires adaptation to different voltage levels, electrical characteristics, timing specifications, and command protocols. Both the value and the challenge of programmable IO are highlighted by the large area devoted to IOs on FPGAs. For example, Altera Stratix II (90 nm) devices devote 20% (largest device) to 48% (smallest device) of their die area to IO-related structures and support 28 different IO standards.

As Fig. 12 shows, FPGAs address the challenges of programmable IO design using a combination of approaches (Tyhach et al. 2004; Qian et al. 2018). First,

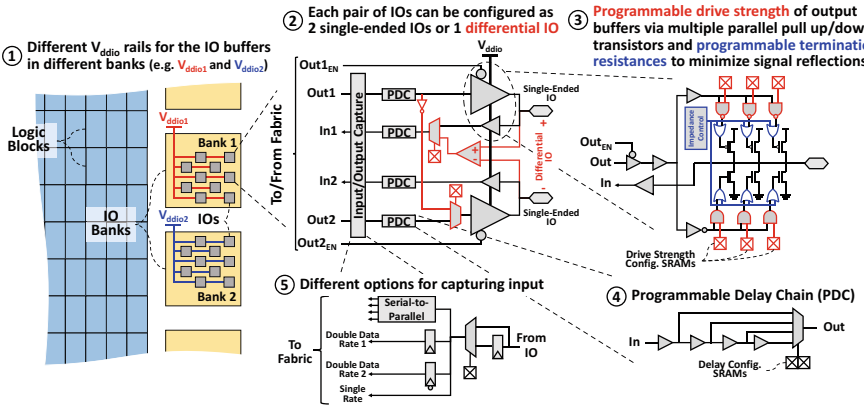

Fig. 12 Overview of the different techniques for implementing programmable IOs in FPGAs

FPGAs use IO buffers that can operate across a range of voltages. As shown in ①, these IOs are grouped into banks (commonly on the order of 50 IOs per bank), where each bank has a separate V_{ddio} rail for the IO buffers. This allows different banks to operate at different voltage levels. For example, IOs in one bank could be operating at 1.8 V while those in a different bank operate at 1.2 V. Second, each IO can be used separately for single-ended standards, or pairs of IOs can be programmed to implement the positive and negative lines for differential IO standards as in ②. Third, IO buffers are implemented with multiple parallel pull-up and pull-down transistors so that their drive strengths can be programmably adjusted by enabling or disabling different numbers of pull-up/pull-down pairs. This is illustrated in part ③ of Fig. 12. By programming some pull-up or pull-down transistors to be enabled even when no output is being driven, FPGA IOs can minimize signal reflections by implementing different on-chip termination resistances. Programmable delay chains, shown in ④, provide a fourth level of configurability, allowing fine delay adjustments of signal timing to and from the IO buffer.

In addition to electrical and timing programmability, FPGA IO blocks contain additional hardened digital circuitry to simplify capturing and transferring IO data to the fabric. Generally, some or all of this hardened circuitry can be bypassed by SRAM-controlled muxes, allowing FPGA users to choose which hardened functions are desirable for a given design and IO protocol. Part ⑤ of Fig. 12 shows a number of common digital logic options on the IO input path: a capture register, double-to-single data rate conversion registers (used with DDR memories), and serial-to-parallel converters to allow transfers to the programmable fabric operating at a lower frequency. Most FPGAs now also contain bypassable blocks that connect to a group of IOs and implement higher-level protocols like DDR memory controllers. Together these approaches allow the general-purpose FPGA IOs to service many different protocols, at speeds up to 3.2 Gb/s.

The highest speed IOs implement serial protocols, such as PCIe and Ethernet, that embed the clock in data transitions and can run at 28 Gb/s or more. To achieve these speeds, FPGAs include a separate group of differential-only IOs that can only be used as serial transceivers and have less voltage and electrical programmability (Upadhyaya et al. 2016). Just as for the general-purpose IOs, these serial IOs have a sequence of high-speed hardened circuits between them and the fabric, some of which can be optionally bypassed to allow end-users to customize the exact interface protocol.

Overall, FPGA IO design is very challenging, due to the dual (and competing) demands to make the IO not only very fast but also programmable. In addition, the rest of the FPGA fabric should also be designed appropriately to keep up with the IO bandwidth; distributing the very high data bandwidths from IO interfaces requires wide *soft* buses to be configured using the programmable routing and logic. This creates additional challenges that will be discussed later in the "System-Level Interconnect: Network-on-Chip" section.

Programmable Clock Distribution Networks

Since FPGA applications are often communicating with many different devices at different speeds, they commonly include many different clock domains. Most of these clocks are generated on-chip by programmable phase-locked loops (PLLs), delay-locked loops (DLLs) and clock data recovery (CDR) circuits. Distributing that many high-speed clocks to all the FFs on the chip using the general programmable routing (discussed in the "Programmable Routing" section) would be extremely challenging for several reasons:

1. Both the programmable routing architecture and the routing CAD algorithms for general signals focus on optimizing delay and wire usage. However, routing clock signals has a different objective: minimizing the clock skew (i.e., balancing the delay) between different endpoints. While specialized low-skew routing CAD algorithms have been devised, they still struggle to create balanced trees in a general programmable interconnect that is not optimized for this case. The difficulty increases for major system clocks, which can have fanouts of hundreds of thousands of registers.
2. The programmable routing wires are optimized for density and speed rather than minimal process variation, and this increases the uncertainty of clocks routed on them, which in turn degrades timing. Another source of increased uncertainty is the capacitive crosstalk between the densely spaced routing wires. A signal (routed on an adjacent wire) toggling at the same time as the clock edge will add significant clock jitter, degrading both setup and hold timing.
3. The very high toggle rate of clocks makes adding extra capacitance to their routing highly undesirable, as it will have a significant power impact. The inefficiency of the general routing wires in creating balanced trees due to both extra switches and suboptimal switch patterns for this case will lead to higher clock capacitance and power consumption.

As a result, FPGAs typically have dedicated interconnect networks for clock distribution, which still have to be flexible enough since the clock domain of each register can vary from one design to another.

Clock networks use routing wires and switch topologies that allow the construction of low-skew networks like *H-trees*. As shown in Fig. 13, these trees have a fractal pattern in the shape of the letter *H* with the signal source at the center and equal delays to reach the four endpoints. These distribution trees minimize clock uncertainty by using wider metal wires with bigger buffers between tree hierarchy levels to minimize process variation and shielded trees to reduce crosstalk-induced jitter. However, an FPGA design can have dozens of clocks, with many of them spanning sub-regions of the chip near the programmable IOs (e.g., a single DDR3 interface typically uses 5–7 different clocks) (Hutton et al. 2005). Pre-fabricating dozens of H-trees that span the entire chip would be one possible clocking architecture, but it would be very expensive, as the lowest level of each of

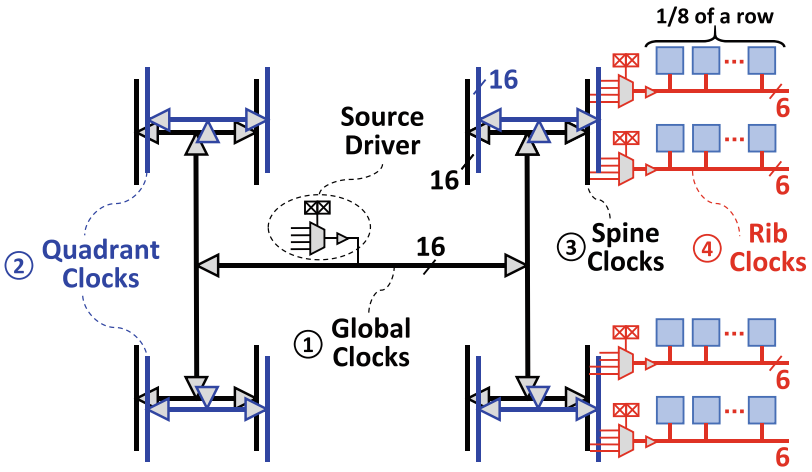

Fig. 13 An example programmable clock distribution network similar to that of Stratix V FPGAs. It has 16 chip-wide global H-trees (black), 16 smaller H-trees per quadrant (blue), and spine-and-ribs leaf distribution (red)

these H-trees would add approximately one (wide and shielded) wire to each routing channel.

Consequently, several techniques are commonly used to implement cheaper clock distribution networks. Since not all clocks are needed everywhere on the chip, some global (chip-wide) H-trees for major clock domains are built along with some smaller ones that cover only portions (e.g., quadrants) of the chip as marked by ① and ② in Fig. 13, respectively. For example, fabricating 16 global and 16 quadrant H-trees enables the use of up to 80 different clocks (16 clocks on the global networks $+ 4 \times 16$ clocks on the quadrant clocks) at a cost equivalent to that of only 32 global H-trees. Additional wire savings are achieved by implementing the leaf wiring in a *spine-and-ribs* style as indicated by ③ and ④ in Fig. 13 instead of continuing the H-tree fractal pattern down to individual blocks. The last wire level in an H-tree is called a *spine clock* and it drives several *rib clocks* that each span a fraction of an FPGA row. The clock skew is tolerable as long as the spine and rib wires are kept reasonably short. To further reduce the cost of the leaf wires (ribs) of the clock network, programmable multiplexers are added to select only a portion of the possible spine clock sources to be routed to the rib clocks that functional blocks can access. In Fig. 13 for example, 32 clock trees are multiplexed down to 6 *rib clocks*, reducing the expensive wiring at the leaves of the clock networks by 81%. This multiplexing leads to a constraint: all the function blocks spanned by a rib clock (1/8 of a row in many Altera/Intel FPGAs) must together use no more than 6 distinct clocks. This constraint is enforced automatically by the placement CAD tool during optimization.

The most recent FPGAs have made clocking networks more flexible. In the Intel Stratix 10 architecture, the FPGA chip is divided into clock sectors where

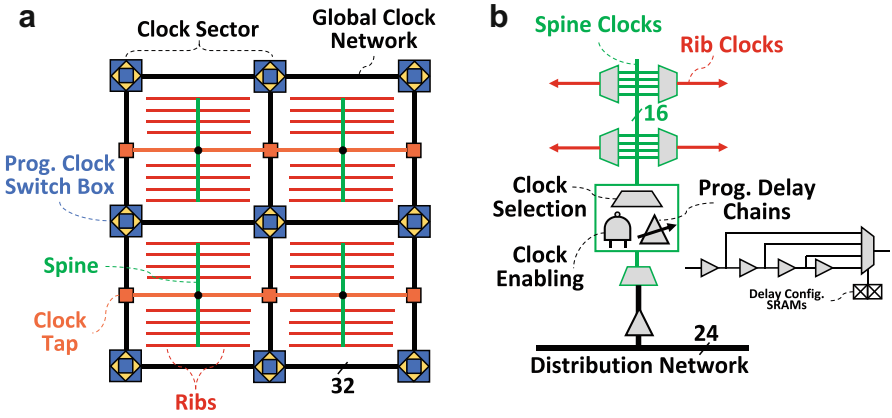

Fig. 14 (**a**) Routable clock networks in Intel Stratix 10 and (**b**) Spine clock control in Xilinx Ultrascale+

traditional spine-and-ribs clock distribution is used within each sector, as shown in Fig. 14a. At the chip level, 32 more flexible distribution networks are implemented with programmable buffers and switch boxes; these networks route each clock to the center of each clock sector that uses it. This highly flexible network can be used to implement a conventional full H-tree, multiple smaller H-trees, or irregular skew-balanced trees as determined by the CAD tools to fit more clocks and minimize clock skew (Ebeling et al. 2016). This clock distribution network has many programmable switches, but unlike conventional programmable routing the switch pattern is optimized for the creation of balanced structures like H-trees and the wires are designed for low process variation and are shielded against crosstalk.

The Ultrascale+ architecture from Xilinx implements clock enable circuitry for power reduction and programmable delay chains for time borrowing at the spine level as illustrated in Fig. 14b. This causes a less than 1% increase in the FPGA die size. When combined with pulse latches, these additional programmable delay chains can increase clock frequency by 5–8%, depending on the available hold margin (Ganusov and Devlin 2016). The Versal architecture leverages these programmable delay chains further by calibrating them on power up to account for process variations across the chip (Gaide et al. 2019). This *adaptive deskewing* technique helps reduce the clock uncertainty and allows the chip to run faster by having narrower guard bands in the timing models.

On-chip Memory

FFs in logic blocks were the first storage elements to be integrated into FPGAs, as described in the "Programmable Logic Blocks" section. However, as FPGA logic

capacity grew, they were used to implement more complex systems which almost always require memory to buffer and re-use data. This motivated more on-chip storage options, since building large RAMs out of registers and LUTs is over 100× less dense than a dedicated SRAM memory array. At the same time, the memory requirements of applications implemented on FPGAs are very diverse, including (but not limited to) small coefficient storage RAMs for FIR filters, large buffers for network packets, caches and register files for processor-like modules, read-only memory for instructions, and FIFOs of myriad sizes to decouple computation modules. This means that there is no single RAM configuration (capacity, word width, number of ports) that can satisfy the needs of all FPGA designs, making it challenging to decide on what kind(s) of RAM blocks should be added to an FPGA such that they are efficient for a broad range of uses. The first FPGA to include hard functional blocks for memory (*block RAMs* or *BRAMs*) was the Altera Flex 10 K in 1995. It included columns of small (2 Kb) BRAMs that connect to the rest of the fabric through the programmable routing. Since then, the capacity and diversity of FPGA on-chip memories have been gradually increasing and it is typical for ∼25% of the area of a modern FPGA to be consumed by BRAM tiles (including their programmable routing) (Tatsumura et al. 2016).

Figure 15 illustrates the organization of an SRAM-based BRAM. An FPGA BRAM consists of a traditional SRAM memory array at its core, with additional *peripheral circuitry* that makes them configurable for different purposes and provides flexible connectivity to the programmable routing. The core memory array consists of a two-dimensional array of SRAM cells to store bits, and a considerable amount of peripheral circuitry to orchestrate access to these cells for read/write operations. To simplify timing of the read and write operations, all modern FPGA BRAMs register all their inputs; they also include output registers, but these are configurable and can be bypassed. During a write operation, the column decoder activates the write drivers (WD), which in turn charge the bitlines (BL and $\overline{BL}$) according to the input data to-be-written to the memory cells. Simultaneously, the row decoder activates the wordline (WL) of the row specified by the input write address, connecting one row of cells to their bitlines so they are overwritten with new data. During a read operation, both the BL and $\overline{BL}$ are pre-charged high and then the row decoder activates the wordline of the row specified by the input read address. The contents of the activated cells cause a slight difference in the voltage between BL and $\overline{BL}$, which is sensed and amplified by the sense amplifier (SA) circuit to produce the output data (Tatsumura et al. 2016).

BRAM capacity, data word width, and number of read/write ports are all key architectural parameters. More capable BRAMs cost more silicon area, so architects must carefully balance BRAM design choices while taking into account the most common use cases in application circuits. For example, the area occupied by the memory cells grows linearly with the capacity of the BRAM, but the area of the peripheral circuitry and the number of routing ports grows sub-linearly. This means that larger BRAMs have lower area per bit, making large on-chip buffers more efficient. On the other hand, if an application requires only small RAMs, much of the capacity of a larger BRAM may be left unused. Similarly, a BRAM with a

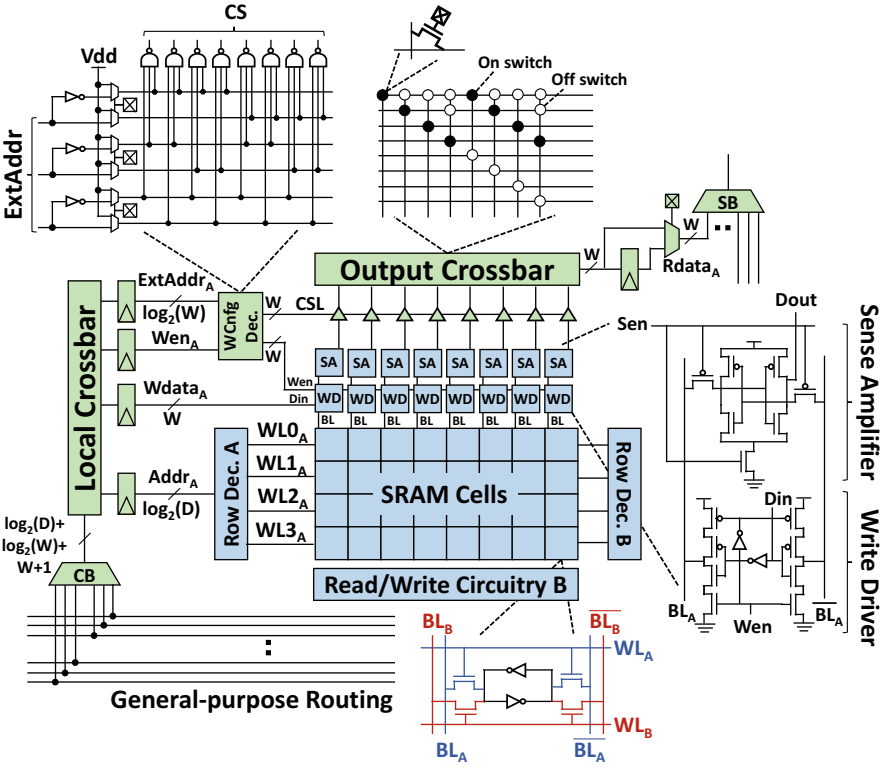

Fig. 15 Organization and circuitry of a conventional dual-port SRAM-based FPGA BRAM. The components highlighted in blue are common in any SRAM-based memory module, while those highlighted in green are FPGA-specific. This BRAM has a maximum data width of 8 bits, but the output crossbar is configured for 4-bit output mode

larger data width can provide higher data bandwidth to downstream logic. However, it costs more area than a BRAM with the same capacity but a smaller word width, as the larger data word width necessitates more sense amplifiers, write drivers and programmable routing ports. Finally, increasing the number of read/write ports to a BRAM increases the area of both the SRAM cells and the peripheral circuitry, but again increases the data bandwidth the BRAM can provide and allows more diverse uses. For example, FIFOs (which are ubiquitous in FPGA designs) require both a read and a write port. The implementation details of a dual-port SRAM cell is shown at the bottom of Fig. 15. Implementing a second port to the SRAM cell (port B highlighted in red) adds two transistors, increasing the area of the SRAM cells by 33%. In addition, the second port also needs an additional copy of the sense amplifiers, write drivers and row decoders (the "Read/Write Circuitry B" and "Row Decoder B" blocks in Fig. 15). If both ports are read/write (r/w), we also have to double the number of ports to the programmable routing.

Because the FPGA on-chip memory must satisfy the needs of *every* application implemented on that FPGA, it is also common to add extra configurability to BRAMs to allow them to adapt to application needs (Wilton et al. 1995). FPGA BRAMs are designed to have configurable width and depth by adding low-cost multiplexing circuitry to the peripherals of the memory array. For example, in Fig. 15 the actual SRAM array is implemented as a 4×8-bit array, meaning it naturally stores 8-bit data words. By adding multiplexers controlled by 3 address bits to the output crossbar, and extra decoding and enabling logic to the read/write circuitry, this RAM can also operate in 8×4-bit, 16×2-bit or 32×1-bit modes. The multipliexers in the width configurability decoder("WCnfg Dec." in Fig. 15) select between V_{dd} and address bits to implement configurable width of between 1 and 8 bits per word for example. The multiplexers are programmed using configuration SRAM cells and are used to generate column select (CS) and write enable (Wen) signals that control the sense amplifiers and write drivers for narrow read and write operations, respectively. For typical BRAM sizes (several Kb or more), the cost of this additional width configurability circuitry is small compared to the cost of a conventional SRAM array and it does not require any additional costly routing ports.

Another unique component of the FPGA BRAMs compared to conventional memory blocks is their interface to the programmable routing fabric. This interface is generally designed to be similar to that of the logic blocks described in the "Programmable Logic Blocks" section; it is easier to create a routing architecture that balances flexibility and cost well if all block types connect to it in similar ways. Connection block multiplexers, followed by local crossbars in some FPGAs, form the BRAM input routing ports, while the read outputs drive switch block multiplexers to form the output routing ports. These routing interfaces are costly, particularly for small BRAMs; they constitute 5% of the area of 256 Kb BRAM tiles, and this portion grows to 35% for smaller 8 Kb BRAMs (Yazdanshenas et al. 2017). This motivates minimizing the number of routing ports to a BRAM as much as possible without unduly comprising its functionality. Table 2 summarizes the number of routing ports required for different numbers and types of BRAM read and write ports. For example, a single-port BRAM (1r/w) requires $W + \log_2(D)$ input ports for write data and read/write address, and W output ports for read data, where W and D are the maximum word width and the BRAM depth, respectively. The table shows that a true dual-port (2r/w) BRAM requires $2W$ more ports compared to a simple dual-port (1r+1w) BRAM, which significantly increases the cost of the routing interfaces. While true dual-port memory is useful for register files, caches

Table 2 Number of routing ports needed for different numbers and types of BRAM read/write ports (W: data width, D: BRAM depth)

BRAM ports	BRAM mode	# Routing ports
1r	Single-port ROM	$\log_2(D) + W$
1r/w	Single-port RAM	$\log_2(D) + 2W$
1r+1w	Simple dual-port RAM	$2\log_2(D) + 2W$
2r/w	True dual-port RAM	$2\log_2(D) + 4W$
2r+2w	Quad-port RAM	$4\log_2(D) + 4W$

and shared memory switches, the most common use of multi-ported RAMs on FPGAs is for FIFOs, which require only one read and one write port (1r+1w rather than 2r/w ports). Consequently, FPGA BRAMs typically have true dual-port SRAM cores but with only enough routing interfaces for simple-dual port mode at the full width supported by the SRAM core (W), and limit the width of the true-dual port mode to only half of the maximum width ($W/2$).

Another way to mitigate the cost of additional BRAM ports is to *multi-pump* the memory blocks by operating the BRAMs at a frequency that is a multiple of that used for the rest of the design logic. By doing so, a physically single-ported SRAM array can implement a logically multi-ported BRAM without the cost of additional ports as in Tabula's *Spacetime* architecture (Halfhill 2010). Multi-pumping can also be used with conventional FPGA BRAMs by building the time-multiplexing logic in the soft fabric (LaForest et al. 2012); however, this leads to aggressive timing constraints for the time-multiplexing logic, which can make timing closure more challenging and increase compile time. For example, Ahmed et al. (2019) showed that careful design partitioning, floorplanning and iterative compilation are necessary for meeting timing on the time-multiplexing logic especially when using a large number of multi-pumped BRAMs. Altera introduced quad-port BRAMs in its Mercury devices in the early 2000s to make shared memory switches (useful in packet processing) and register files more efficent. However, this feature increased the BRAM size and was not sufficiently used to justify its inclusion in subsequent FPGA generations. Instead designers use a variety of techniques to combine dual-ported FPGA BRAMs and soft logic to make highly-ported structures when needed, albeit at lower efficiency (LaForest et al. 2012). We refer the interested reader to both Tatsumura et al. (2016) and Yazdanshenas et al. (2017) for extensive details about the design of BRAM core and peripheral circuitry.

In addtition to BRAMs, most FPGAs can re-use at least some of their LUTs as memory. The truth tables in the logic block K-LUTs are $2^K \times 1$-bit read-only memories; they are written once by the configuration circuitry when the design bitstream is loaded. Since LUTs already have read circuitry (read out a stored value based on a K-bit input/address), they can be used as small LUT-based RAMs (LUT-RAMs) just by adding low-cost designer-controlled write circuitry. However, a major concern is the number of additional routing ports necessary to implement the write functionality to change a LUT to a LUT-RAM. For example, an ALM in recent Altera/Intel architectures is a 6-LUT that can be fractured into two 5-LUTs and has 8 input routing ports, as explained in the "Programmable Logic Blocks" section. This means it can operate as a 64×1-bit or a 32×2-bit memory with 6 or 5 bits for read address, respectively. This leaves only 2 or 3 unused routing ports, which are not enough for write address, data, and write enable (8 total signals) if we want to read and write in each cycle (simple dual-port mode), which is the most commonly used RAM mode in FPGA designs. To overcome this problem, an entire logic block of 10 ALMs is configured as a LUT-RAM to amortize the control circuitry and address bits across 10 ALMs. The write address and write enable signals are assembled by stealing a single unused routing port

from each ALM and broadcasting the resulting address and enable to all the ALMs in a logic block (Lewis et al. 2009). Consequently, a logic block can implement a 64×10-bit or 32×20-bit simple dual-port RAM, but has a restriction that a single logic block cannot mix logic and LUT-RAM. Xilinx Ultrascale similarly converts an entire logic block to LUT-RAM, but all the routing ports of one out of the eight LUTs in a logic block are repurposed to drive the shared write address and enable signals. Therefore, a Xilinx logic block can implement a 64×7-bit or 32×14-bit simple dual-port RAM, or a slightly wider single-port RAM (64×8-bit or 32×16-bit). Avoiding extra routing ports keeps the cost of LUT-RAM low, but it still adds some area. Since it would be very unusual for a design to use more than 50% of the logic fabric as LUT-RAMs, both Altera/Intel and Xilinx have elected to make only half (or less) of their logic blocks LUT-RAM capable in their recent architectures, thereby further reducing the area cost.

Designers require many different RAMs in a typical design, all of which must be implemented by the fixed BRAM and LUT-RAM resources on the chip. Forcing designers to determine the best way to combine BRAM and LUT-RAM for each memory configuration they need and writing Verilog to implement them would be laborious and would also impede migration of the design to a new FPGA architecture. Instead, the vendor CAD tools include a *RAM mapping* stage that implements the *logical* memories in the user's design using the *physical* BRAMs and LUT-RAMs on the chip. The RAM mapper chooses the physical memory implementation (i.e., memory type and the width/number/type of its ports) and generates any additional logic required to combine multiple BRAMs or LUT-RAMs to implement each logical RAM. An example of mapping a logical 2048×32-bit RAM with 2 read and 1 write ports to an FPGA with physical 1024×8-bit dual-port BRAMs is illustrated in Fig. 16. First, four physical BRAMs are combined in parallel to make wider RAMs with no extra logic. Then, soft logic resources are used to perform depth-wise stitching of two sets of four physical BRAMs, such that

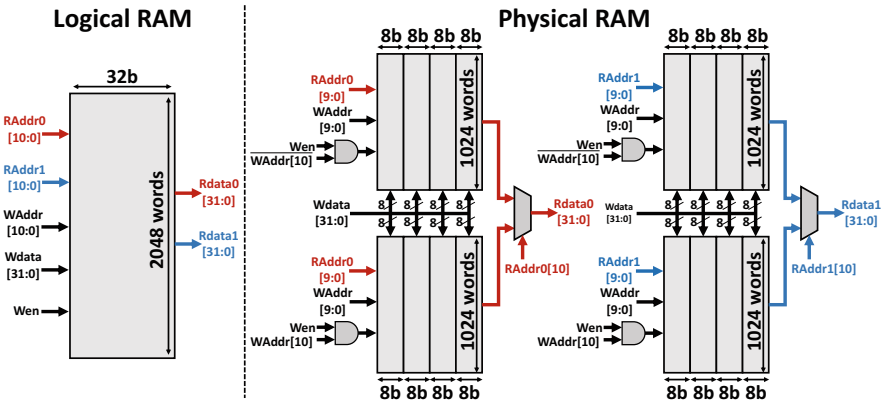

Fig. 16 Mapping a 2048×32-bit 2r+1w logical RAM to an FPGA with 1024×8-bit 1r+1w physical BRAMs

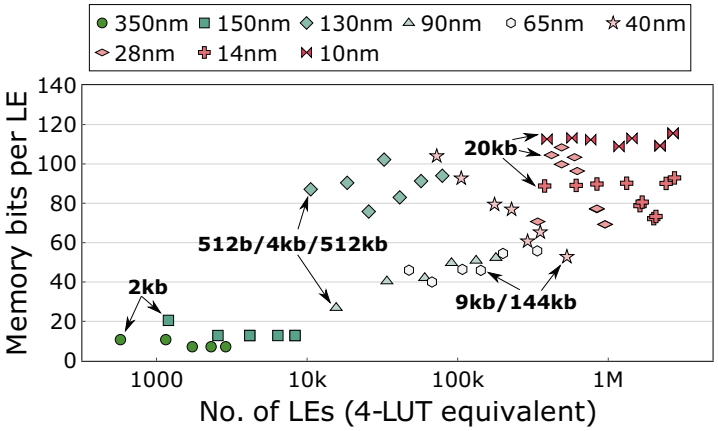

Fig. 17 Memory bits per logic elements for different generations of Altera/Intel FPGAs starting from the 350 nm Flex 10K (1995) to the 10 nm Agilex (2019) architecture. FPGA on-chip memory density has increased by a factor of 16× in the last 25 years. The labels show the sizes of BRAMs in each generation

the most-significant bits of the write and read addresses are used as write enable and read output multiplexer select signals, respectively. Finally, in this case, we require two read ports and one write port while the physical BRAMs only support a maximum of 2r/w ports. To implement the second read port, the whole structure is either replicated as shown in the figure or double-pumped as previously explained. Several algorithms for optimizing RAM mapping are described in Tessier et al. (2007) and Lai and Lin (2016).

Over the past 25 years, FPGA memory architecture has evolved considerably and has also become increasingly important, as the ratio of memory to logic on an FPGA die has grown significantly. Figure 17 plots the memory bits per logic element (including LUT-RAM) versus the number of logic elements in Altera/Intel devices starting from the 350 nm Flex 10K devices (1995) to 10 nm Agilex devices (2019). There has been a gradual increase in the memory richness of FPGAs over time, and to meet the demand for more bits at a cheaper cost, modern BRAMs have larger capacities (20 Kb) than the first BRAMs (2 Kb). Some FPGAs have had highly heterogeneous BRAM architectures in order to provide some physical RAMs that are efficient for small or wide logical RAMs, and others that are efficient for large and relatively narrow logical RAMs. For example, Stratix (130 nm) had 3 types of BRAM, with capacities of 512 b, 4 Kb and 512 Kb. The introduction of LUT-RAM in Stratix III (65 nm) reduced the need for small BRAMs, so it moved to a memory architecture with only medium and large size (9 Kb and 144 Kb) BRAMs. Stratix V (28 nm) and later Intel devices have moved to a combination of LUT-RAM and a single medium-sized BRAM (20 Kb) to simplify both the FPGA layout as well as RAM mapping and placement. A similar trend can be observed in Xilinx devices (Tatsumura et al. 2016); Xilinx's RAM architecture also combines LUT-RAM and a

medium-sized 18 Kb RAM, but also includes hard circuitry to combine two BRAMs into a single 36 Kb block. However, Xilinx's most recent devices add a large 288 Kb BRAM (UltraRAM) to be more efficient for very large buffers, showing that there is still no general agreement on the best BRAM architecture. Some recent Intel devices further enhance their memory capacity by integrating the FPGA fabric with one or more embedded SRAM (eSRAM) chiplets using interposer technology that will be discussed in the "Interposers" section later. Each eSRAM chiplet implements eight large simple dual-port memories with a combined capacity of 47 Mb in Stratix 10 and 18 Mb in Agilex. These memories are ideal for wide and deep buffers that exceed on-chip storage capacity, but benefit from reduced latency; for example, routing tables or packet headers in networking applications.

To give some insight into the relative areas and efficiencies of different BRAMs, Table 3 shows the resource usage, silicon area, and frequency of a 2048×72-bit logical RAM when it is implemented by Quartus (the CAD flow for Altera/Intel FPGAs) in a variety of ways on a Stratix IV device. The silicon areas are computed using the published Stratix III block areas from Wong et al. (2011) and scaling them from 65 nm down to 40 nm, as Stratix III and IV have the same architecture but use different process nodes. As this logical RAM is a perfect fit to the 144 Kb BRAM in Stratix IV, it achieves the best area when mapped to a single 144 Kb BRAM. Interestingly, mapping to eighteen 9 Kb BRAMs is only 1.9× larger in silicon area (note that output width limitations lead to 18 BRAMs instead of the 16 one might expect). The 9 Kb BRAM implementation is actually faster than the 144 Kb BRAM implementation, as the smaller BRAMs have higher maximum operating frequencies. Mapping such a large logical RAM to LUT-RAMs is inefficient, requiring 12.7× more area and running at 40% of the frequency. Finally, mapping only to the logic and routing resources highlights the importance of BRAMs; the area is over 300× larger than the 144 Kb BRAM implementation. While the 144 Kb BRAM is most efficient for this single test case, real designs have diverse logical RAMs, and for small or shallow memories the 9 Kb and LUT-RAM options would outperform the 144 Kb BRAM, motivating a diversity of on-chip RAM resources. To choose the best mix of BRAM sizes and maximum word widths, one needs both a RAM mapping tool and tools to estimate the area, speed and power of each BRAM (Yazdanshenas et al. 2017). Published studies into BRAM architecture trade-offs for FPGAs include (Yazdanshenas et al. 2017; Lewis et al. 2013).

Until now, all commercial FPGAs use only SRAM-based memory cells in their BRAMs. With the desire for more dense BRAMs that would enable more memory-

Table 3 Implementation results for a 2048×72-bit 1r+1w RAM using BRAMs, LUT-RAMs and registers on Stratix IV

Implementation	Half-ALMs	BRAMs		Area (mm^2)	Freq. (MHz)
		9K	144K		
144K BRAMs	0	0	1	0.22 (1.0×)	336 (1.0×)
9K BRAMs	0	18	0	0.41 (1.9×)	497 (1.5×)
LUT-RAM	6597	0	0	2.81 (12.8×)	134 (0.4×)
Registers	165155	0	0	68.8 (313×)	129 (0.4×)

rich FPGAs and SRAM scaling becoming increasingly difficult due to process variation, a few academic studies have explored the use of other emerging memory technologies such as magnetic tunnel junctions (MTJs) to build FPGA memory blocks. According to Tatsumura et al. (2016), MTJ-based BRAMs could increase the FPGA memory capacity by up to 2.95× with the same die size; however, they would increase the process complexity.

DSP Blocks

Initially, the only dedicated arithmetic circuits in commercial FPGA architectures were carry chains to implement efficient adders, as discussed in the "Programmable Logic Blocks" section earlier. Thus, multipliers had to be implemented in the soft logic using a combination of LUTs and carry chains, which for larger operand bit widths incurs significant logic utilization and delay. As wireless communication and signal processing became major FPGA markets, system designers proposed novel implementations to mitigate the inefficiency of multiplier implementations in soft logic. For example, the multiplier-less *distributed arithmetic* technique was proposed to implement efficient FIR filter structures in LUTs (Meher et al. 2008).

With the prevalence of multipliers in FPGA designs from key application domains and their lower efficiency when implemented in soft logic, they quickly became a candidate for hardening as dedicated circuits in FPGA architectures. An N-bit multiplier array consists of N^2 logic gates to generate partial products and compression trees to reduce them, with only $2N$ inputs and $2N$ outputs. Therefore, the high gains of hardening the multiplier logic and the relatively low cost of the programmable interfaces to the FPGA's routing fabric strongly advocated for adopting hard multipliers in subsequent FPGA architectures. As shown in the top left of Fig. 18, Xilinx introduced its Virtex-II architecture with the industry's first 18×18 bit hard multiplier blocks. To simplify the layout integration with the full-custom FPGA fabric, these multipliers were arranged in columns right beside BRAM columns. In order to further reduce the interconnect cost, the multiplier block and its adjacent BRAM had to share some interconnect resources, limiting the maximum usable data width of the BRAM block when the multiplier is used for computation. Multiple hard 18-bit multipliers could be stitched together with soft logic to form bigger multipliers or FIR filters.

In 2002, Altera adopted a different approach by introducing more fully-featured DSP blocks targeting the communications and signal processing domains in their Stratix architecture (Lewis et al. 2003) (see the second block in Fig. 18). The main design philosophy of this DSP block was to minimize the amount of soft logic resources used to implement common DSP algorithms by hardening more functionality inside the DSP block and enhancing its flexibility to allow more applications to use it. The Stratix DSP block was highly configurable with support for different modes of operation and multiplication precisions unlike the fixed-function 18-bit multipliers in Virtex-II. Each Stratix variable-precision DSP block spanned 8 FPGA rows and could implement eight 9×9 bit multipliers, four 18×18 bit multipliers, or one 36×36 multiplier.

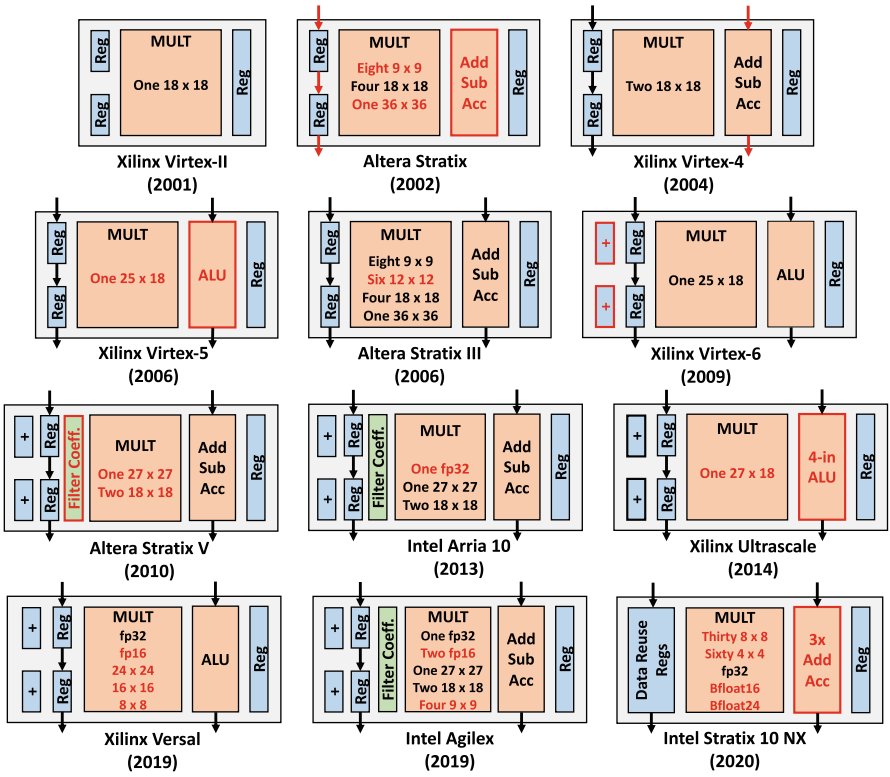

Fig. 18 DSP block evolution in Altera/Intel and Xilinx FPGAs. Incrementally added features are highlighted in red

These modes of operation selected by Altera highlight an important theme of designing FPGA hard blocks: increasing the flexibility and utility of these blocks by adding low-cost circuitry such that it becomes more broadly useful. For example, an 18×18 multiplier array can be decomposed into two 9×9 arrays that together use the same number of inputs and outputs (and hence routing ports). Similarly, four 18×18 multipliers can be combined into one 36×36 array using cheap glue logic. Figure 19 shows how an 18×18 multiplier array can be fractured into multiple 9×9 arrays. It can be split into four 9×9 arrays by doubling the number of input and output pins. However, to avoid adding these costly routing interfaces, two of the four 9×9 arrays are left unused (grey circles) and the other two (blue circles) are used to perform the two multiplications $A0 \times B0$ and $A1 \times B1$. This is done by splitting the partial product compressor trees at the positions indicated by the red dashed lines and adding inverting capabilities to the border cells of the top-right array, marked with crosses in Fig. 19 to implement two's complement signed multiplication using the Baugh-Wooley algorithm (the bottom left array already has the inverting capability from the 18×18 array).

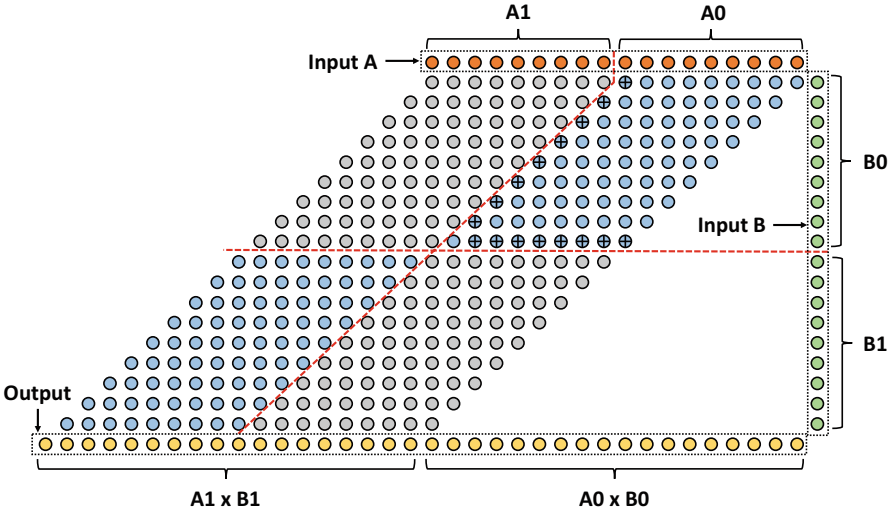

Fig. 19 Fracturing an 18×18 multiplier array into two 9×9 arrays with the same number of input/output ports

In addition to the fracturable multiplier arrays, the Stratix DSP also incorporated an adder/output block to perform summation and accumulation operations, as well as hardened input registers that could be configured as shift registers with dedicated cascade interconnect between them to implement efficient FIR filter structures. Xilinx also adopted a fully-featured DSP block approach by introducing their *DSP48* tiles in the Virtex-4 architecture. Each DSP tile had two fixed-precision 18×18 bit multipliers with similar functionalities to the Stratix DSP block (e.g., input cascades, adder/subtractor/accumulator). Virtex-4 also introduced the ability to cascade the adders/accumulators using dedicated interconnects on the output side of the DSP blocks to implement high-speed systolic FIR filters with hardened reduction chains.

An N-tap FIR filter performs a discrete 1D convolution between the samples of a signal $X = \{x_0, x_1, \ldots, x_T\}$ and certain coefficients $C = \{c_0, c_1, \ldots, c_{N-1}\}$ that represent the impulse response of the desired filter, as shown in Eq. (1).

$$y_n = c_0 x_n + c_1 x_{n-1} + \ldots + c_N x_{n-N} = \sum_{i=0}^{N} c_i x_{n-i} \tag{1}$$

Many of the FIR filters used in practice are symmetric with $c_i = c_{N-i}$, for $i = 0$ to $N/2$. As a result of this symmetry, the filter computation can be refactored as shown in Eq. (2).

$$y_n = c_0[x_n + x_{n-N}] + \ldots + c_{N/2-1}[x_{n-N/2-1} + x_{n-N/2}] \tag{2}$$

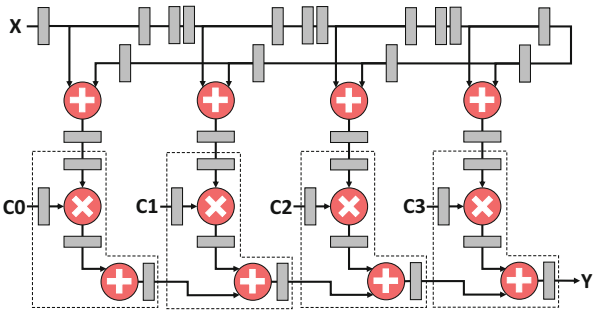

Fig. 20 Systolic implementation of a symmetric FIR filter circuit

Figure 20 shows the structure of a systolic symmetric FIR filter circuit, which is a key use case for FPGAs in wireless base stations. Both Stratix and Virtex-4 DSP blocks can implement the portions highlighted by the dotted boxes, resulting in significant efficiency gains compared to implementing them in the FPGA's soft logic. Interestingly, while FPGA CAD tools will automatically implement a multiplication operation (written as a * operator in RTL) in DSP blocks, they will generally not make use of any of the advanced DSP block features (e.g., accumulation, systolic registers for FIR filters) unless a designer manually instantiates a vendor-supplied DSP block IP in the proper mode. Consequently, using the more powerful DSP block features makes a design less portable when migrating to another FPGA with different DSP block capabilities. Some work has extended automatic DSP block inference to sequences of multiply, add and subtract operations in RTL that exactly match the DSP block capabities (Ronak and Fahmy 2015a). This can improve automatic inference to some extent, but it will be difficult to extend to fully utilize advanced DSP block features like coefficient re-use networks.

The Stratix III/IV DSP block was similar to the Stratix II one but could implement four 18×18 multipliers per half a DSP block (instead of two) if their results are summed to limit the number of output routing interfaces. Table 4 lists the implementation results of both symmetric and asymmetric 51-tap 16-bit FIR filters, with and without using the hard DSP blocks on a Stratix IV device. When DSP blocks are not used, we experiment with two different cases: fixed filter coefficients, and filter coefficients that can change at runtime. If the filter coefficients are fixed, the multiplier arrays implemented in the soft logic are optimized by synthesizing away parts of the partial product generation logic that correspond to zero bits in the coefficient values. Hence, it has lower resource utilization than with input coefficients that can change at runtime. For the symmetric filter, even when using the DSP blocks, we still need to use some soft logic resources to implement the input cascade chains and pre-adders, as shown in Fig. 20. Using the hard DSP blocks results in 3× higher area efficiency vs. using the soft fabric in the case of fixed coefficients. This gap grows to 6.2× for filter coefficients that are changeable during runtime. For the asymmetric filter, the complete FIR filter structure can be implemented in the DSP blocks without any soft logic resources. Thus, the

Table 4 Implementation results for a 51-tap 16-bit FIR filter on Stratix IV with and without using the hardened DSP blocks

Symmetric Filter				
Implementation	Half-ALMs	DSPs	Area (mm^2)	Freq. (MHz)
With DSPs	403	$3\frac{2}{8}$	0.49 (1.0×)	510 (1.0×)
Without DSPs (fixed coeff.)	3505	0	1.46 (3.0×)	248 (0.5×)
Without DSPs (variable coeff.)	7238	0	3.01 (6.2×)	220 (0.4×)
Asymmetric Filter				
Implementation	half-ALMs	DSPs	Area (mm^2)	Freq. (MHz)
With DSPs	0	$6\frac{3}{8}$	0.63 (1.0×)	510 (1.0×)
Without DSPs (fixed coeff.)	5975	0	2.48 (3.9×)	245 (0.5×)
Without DSPs (variable coeff.)	12867	0	5.35 (8.5×)	217 (0.4×)

area efficiency gap increases to 3.9× and 8.5× for fixed and variable coefficients, respectively. These gains are large but still less than the 35× gap between FPGAs and ASICs (Kuon and Rose 2007) usually cited in academia. The difference is partly due to some soft logic remaining in most application circuits, but even in the case where the FIR filter perfectly fits into DSP blocks with no soft logic, the area reduction hits a maximum of 8.5×. The primary reasons for the lower than 35× gain of Kuon and Rose (2007) are the interfaces to the programmable routing and the general inter-tile programmable routing wires and muxes that must be implemented in the DSP tile. In all cases, using the hard DSP blocks results in about 2× frequency improvement as shown in Table 4. Similarly to BRAMs, the high operating frequencies of DSP blocks mean they can often be multi-pumped (run at a multiple of the soft logic frequency); this is mainly used for resource reduction in DSP-bound designs as in Ronak and Fahmy (2015b).

The next few FPGA architecture generations from both Altera and Xilinx witnessed only minor changes in the DSP block architecture. The main focus of both vendors was to fine-tune the DSP block capabilities for emerging application domains without adding costly programmable routing interfaces. In Stratix V, the DSP block was greatly simplified to natively support two 18×18 bit multiplications (suitable for signal processing) or one 27×27 multiplication (suitable for single-precision floating-point mantissa multiplication). As a result, the simpler Stratix V DSP block spanned a single row, which is more friendly to Altera's row redundancy scheme (i.e., the ability to skip single FPGA rows with fabrication faults in them to increase the effective yield). In addition, input pre-adders as well as embedded coefficient banks to store read-only filter weights were added, which allowed implementation of the whole symmetric FIR filter structure shown in Fig. 20 inside the DSP blocks without the need for any soft logic resources. Xilinx followed a similar path in incorporating 27×18 multiplication with support for pre-adders in Virtex-6 DSP blocks.

As shown in Fig. 18, Xilinx DSP blocks since Virtex-5 have incorporated an ALU that can perform logic operations as well as add and subtract; both the ALU operation and the data paths through the DSP are selected by additional inputs so they can change dynamically from cycle to cycle. This enhancement makes these DSP blocks well suited for the datapath of a soft processor (Cheah et al. 2014). Controlling DSP operations dynamically in this manner increases the flexibility of the block, but has some area cost as adding routing input ports for dynamic control signals is more expensive than adding configuration SRAM cells to statically select operations.

As illustrated in Fig. 18, up to 2009 the evolution of the DSP block architecture was mainly driven by the precisions and requirements of communication applications, especially in wireless base stations, with very few academic research explorations. With the large-scale deployment of FPGAs in datacenters and the emergence of DL as a key component of many applications both in datacenter and edge workloads, the DSP block architecture has evolved in two different directions. The first direction targets the high-performance computing (HPC) domain by adding native support for single-precision floating-point (`fp32`) multiplication. Before that, FPGA vendors would supply designers with IP cores that implement floating-point arithmetic out of fixed-point DSPs and a considerable amount of soft logic resources. This created a major barrier for FPGAs to compete with CPUs and GPUs (which have dedicated floating-point units) in the HPC domain. Native floating-point capabilities were first introduced in Intel's Arria 10 architecture, with a key design goal of avoiding a large increase in DSP block area (Langhammer and Pasca 2015). By reusing the same interface to the programmable routing, not supporting uncommon features like subnormals, flags and multiple rounding schemes, and maximizing the reuse of existing fixed-point hardware, the block area increase was limited to only 10% (which translates to 0.5% total die area increase). Floating-point capabilities are supported in all subsequent generations of Intel FPGAs and in the *DSP58* tiles of the Xilinx Versal architecture (Gaide et al. 2019).

The second direction targets increasing the density of low-precision integer multiplication specifically for DL inference workloads. Prior work has demonstrated the use of low-precision fixed-point arithmetic (8-bit and below) instead of `fp32` at negligible or no accuracy degradation, but greatly reduced hardware cost (Wang et al. 2019). However, the required precision is model-dependent and can even vary between different layers of the same model. As a result, FPGAs have emerged as an attractive solution for DL inference due to their ability to implement custom precision datapaths. This has led both academic researchers and FPGA vendors to investigate adding native support for low-precision multiplication to DSP blocks. Boutros et al. (2018) enhanced the fracturability of an Intel-like DSP block to support more `int9` and `int4` multiply and MAC operations, while keeping the same DSP block routing interface and ensuring its backward compatibility. The proposed DSP block could implement four `int9` and eight `int4` multiply/MAC operations along with Arria-10-like DSP block functionality at the cost of 12% DSP block area increase, which is equivalent to only 0.6% increase in total die area. This DSP block increased the performance of 8-bit and 4-bit DL accelerators

by 1.3× and 1.6× while reducing the utilized FPGA resources by 15% and 30% respectively, compared to an FPGA with DSPs that do not natively support these modes of operation. Another academic work (Rasoulinezhad et al. 2019) enhanced a Xilinx-like DSP block by including a fracturable multiplier array instead of the fixed-precision multiplier in the *DSP48E2* block to support `int9`, `int4` and `int2` precisions. It also added a FIFO register file and special dedicated interconnect between DSP blocks to enable more efficient standard, point-wise and depth-wise convolution layers. Shortly after, the Intel Agilex DSP block added support for an `int9` mode of operation along with half-precision floating-point (`fp16`) and brain float (`bfloat16`) precisions as well. Also, the Xilinx Versal architecture now natively supports `int8` multiplications in its *DSP58* tiles (Gaide et al. 2019).

Throughout the years, the DSP block architecture has evolved to best suit the requirements of key application domains of FPGAs, and provide higher flexibility such that many different applications can benefit from its capabilities. The common focus across all the steps of this evolution was reusing multiplier arrays and routing ports as much as possible to best utilize both these costly resources. However, this becomes harder with the recent divergence in the DSP block requirements of key FPGA application domains between high-precision floating-point in HPC, medium-precision fixed-point in communications, and low-precision fixed-point in DL. As a result, Intel introduced its first domain-specialized FPGA optimized for artificial intelligence (AI) workloads, the Stratix 10 NX. This new FPGA replaces conventional DSP blocks with AI tensor blocks (Langhammer et al. 2021). The tensor blocks drop the support for legacy DSP modes and precisions that were targeting the communications domain and adopt new ones targeting the DL domain specifically. This tensor block significantly increases the number of `int8` and `int4` MACs to 30 and 60 per block respectively, at almost the same die size. Feeding all multipliers with inputs without adding more routing ports is a key concern. Accordingly, the NX tensor block introduces a double-buffered data reuse register network that can be sequentially loaded from a smaller number of routing ports, while allowing common DL compute patterns to make the best use of all available multipliers. Recent work has shown that the Stratix 10 NX with tensor blocks can deliver an average 3.5× performance boost compared to FPGAs with conventional DSP blocks for real-time DL inference workloads (Boutros et al. 2020).

Processor Subsystems

As the complexity of FPGA applications increased, many designs required a software-programmable processor for lightweight control, housekeeping, performance monitoring, or debugging. Therefore, designers had to build and optimize their own *soft processors* out of the FPGA's programmable function blocks and routing (as shown in Fig. 21a), which was a significantly laborious and challenging task. To facilitate the integration of processor subsystems in FPGA designs, FPGA vendors supplied heavily optimized and parameterized soft processor IPs that designers can readily use such as the Nios and Microblaze soft processors from

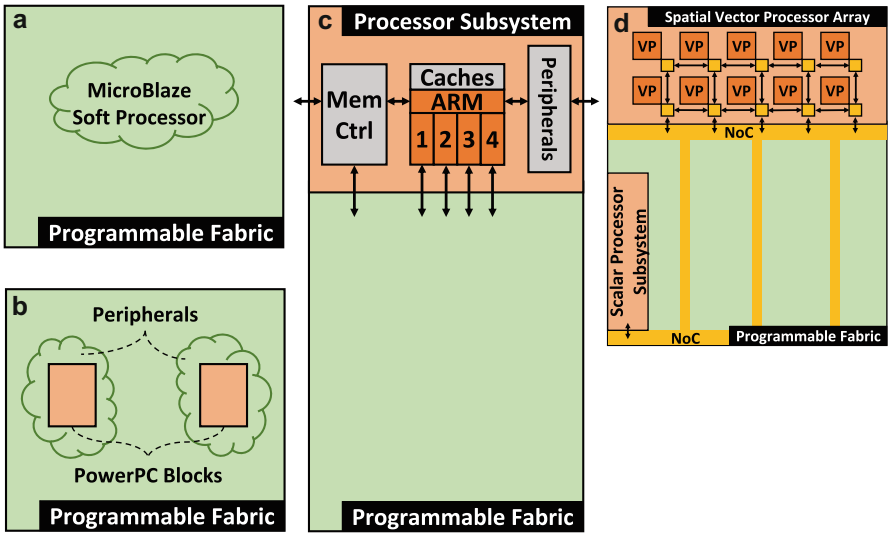

Fig. 21 (**a**) Early Xilinx FPGA with a MicroBlaze soft processor implemented in soft logic, (**b**) Xilinx Virtex-II Pro FPGA with 2 hard PowerPC blocks whose peripherals are implemented in soft logic, (**c**) Xilinx Zynq Ultrascale+ with a complete hard processor subsystem, and (**d**) Xilinx Versal architecture with both a hard scalar processor subsystem and a spatial vector processor array

Altera and Xilinx, respectively. This alleviated the design burden from FPGA users while still allowing them to flexibly configure their architecture parameters (e.g., instruction/data cache sizes, number of cache levels, ALU capabilities, etc.) to match the application requirements. However, these soft processors are still area-inefficient, slower, and have limited capabilities (e.g., scalar, single-issue, in-order microarchitecture) compared to mainstream CPUs, which makes them more suitable for lightweight control and housekeeping tasks rather than compute-oriented ones. The gap is even larger compared to direct hardware execution on repetitive tasks. For example, a Nios II soft processor on a Stratix IV FPGA runs at 250 MHz and consumes 1130 LUTs, 4 DSPs, and 11 BRAMs. When used to compute a simple third-degree polynomial, it has 50× less performance, 130× higher energy, and 2× higher LUT utilization compared to a dedicated hardware implementation (configured into the FPGA) of the same function. Some studies attempt to optimize scalar soft processors for more compute-intensive tasks by adding support for vector instructions. Yiannacouras et al. show that a vector soft processor can improve performance by 25× over a scalar soft processor; while area increases, the area-delay product is still 3× better than a scalar soft processor (Yiannacouras et al. 2009).

As more systems incorporated processors for control and less compute-intensive tasks, FPGA vendors began to harden processor cores to increase performance vs. soft processors. For example, the Xilinx Virtex-II Pro architecture had up to

2 IBM PowerPC RISC processor blocks as illustrated in Fig. 21b, while Altera integrated an ARM core in the Apex architecture. These initial efforts hardened only the *raw* processor core with primitive wire interfaces to the programmable fabric, while the rest of the processor subsystem (e.g., memory controller and peripherals) had to be implemented in the soft logic. This was still time-consuming and did not show enough efficiency gains compared to soft processors to justify the higher design effort and reduced configurability; consequently, these hardened processor-core-only systems were not very successful. With FPGAs growing into more complex and heterogeneous platforms, complete hard processor subsystems (i.e., processors along with their key peripherals) have been incorporated in recent FPGA architectures. This approach has been much more successful as it provides designers with an easy-to-use software environment for implementing portions of their applications, while still achieving a significantly higher performance and energy efficiency compared to soft processors. Consequently, high-performance full-featured hard processor subsystems are now available in most FPGA families. For example, Xilinx's Zynq Ultrascale+ (in Fig. 21c) has an embedded quad-core ARM Cortex-A53 processor along with a cache coherency unit, a memory management unit, direct memory access controller, and many different IO peripherals (e.g., USB, I^2C, UARTs, GPIOs, etc.) to communicate with the outside world, as well as the tightly coupled FPGA fabric. These hybrid devices can be used in many applications where the processor handles strictly serial and branching portions of the workload while the highly-parallel compute-intensive portions are offloaded to the FPGA – this echoes the initial vision for reconfigurable computer architectures in the 1960s (Estrin 1960).

The Xilinx Versal architecture integrates not only an FPGA fabric and a traditional hard processor subsystem, but also a many-core vector processor complex with bus-based reconfigurable interconnect, as shown in Fig. 21d. This architecture still has a spatial nature (similar to an FPGA), and combines the software-level programmability of vector processors with the flexibility of programmable interconnects, making processor cores essentially another form of logic blocks in reconfigurable devices. This new architecture is initially targeted at 5G signal processing and AI, two large and compute-intensive markets for FPGAs. New tools for architecture exploration and evaluation of these highly heterogeneous devices are also emerging (Boutros et al. 2022), enabling new research into both their programming models and efficiency in various applications.

System-Level Interconnect: Network-on-Chip

As FPGAs have grown in both capacity and IO speed, distributing ever higher bandwidth data streams throughout an ever larger fabric has become challenging. Traditionally the *system-level interconnect* that connects high-speed IO interfaces such as DDR, PCIe and Ethernet to modules implemented in the FPGA fabric has been implemented as *soft* buses. These soft buses include multiplexing, arbitration, pipelining and wiring between the relevant endpoints. As the data bandwidth of

external IO interfaces has increased, these soft buses have been forced to become very wide to carry the larger data streams, increasing their resource utilization and making timing closure harder. For example, a single channel of high-bandwidth memory (HBM) has a 128-bit double data rate interface operating at 1 GHz, so a bandwidth-matched soft bus running at 250 MHz must be 1024 bits wide. With recent FPGAs incorporating up to 8 HBM channels as well as numerous PCIe, Ethernet and other interfaces, system level interconnect can rapidly use a major fraction of the FPGA logic and routing resources. In addition, system-level interconnect tends to span long distances. The combination of very wide and physically long buses makes timing closure challenging and usually requires deep pipelining of the soft bus, further increasing its resource use. The system-level interconnect challenge is becoming more difficult in advanced process nodes, as the number and speed of FPGA external interfaces increases, and the metal wire parasitics (and thus interconnect delay) scales poorly (Bohr 1995).

Abdelfattah and Betz (2013) proposed embedding a hard, packet-switched network-on-chip (NoC) in the FPGA fabric to enable more efficient and easier-to-use system-level interconnect. Although a full-featured packet-switched NoC could be implemented using the soft logic and routing of an FPGA, an NoC with hardened routers and links is 23× more area efficient, 6× faster, and consumes 11× less power compared to a *soft* NoC. Designing a hard NoC for an FPGA is challenging since the FPGA architect must commit many choices to silicon (e.g., number of routers, link width, NoC topology) yet still maintain the flexibility of an FPGA to implement a wide variety of applications using many different external interfaces and communication endpoints. Work in Abdelfattah and Betz (2013) advocates for a mesh topology with a moderate number of routers (e.g., 16) and fairly wide (128-bit) links; these choices keep the area cost to less than 2% of the FPGA while ensuring the NoC is easier to lay out and a single NoC link can carry the entire bandwidth of a DDR channel. A hard NoC must also be able to flexibly connect to user logic implemented in the FPGA fabric. Abdelfattah et al. (2015) introduced the *fabric port* which interfaces the hard NoC routers to the FPGA programmable fabric by performing width adaptation, clock domain crossing and voltage translation. This decouples the NoC from the FPGA fabric such that the NoC can run at a fixed (high) frequency, and still interface to FPGA logic and IO interfaces of different speeds and bandwidth requirements with very little glue logic. Hard NoCs also appear very well suited to FPGAs in datacenters. Datacenter FPGAs are normally configured in two parts: a *shell* provides system-level interconnect to the external interfaces, and a *role* implements the application acceleration functionality (Caulfield et al. 2016). The resource use of the shell can be significant: it requires 23% of the device resources in the first generation of Microsoft's Catapult systems (Putnam et al. 2014). Yazdanshenas and Betz (2018) showed that a hard NoC significantly improves resource utilization, operating frequency and routing congestion in datacenter FPGAs. Other studies have proposed FPGA-specific optimizations to increase the area efficiency and performance of soft NoCs (Kapre and Gray 2017; Papamichael and Hoe 2012). However, Yazdanshenas

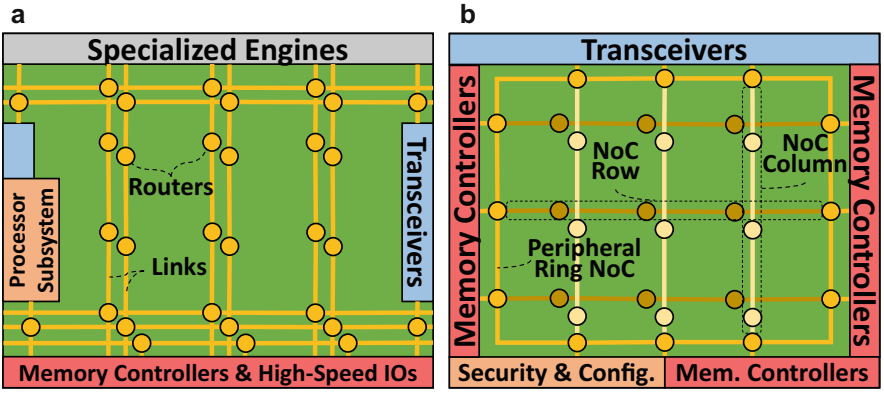

Fig. 22 Network-on-Chip system-level interconnect in next-generation (**a**) Xilinx Versal and (**b**) Achronix Speedster7t architectures

and Betz (2018) showed that even optimized soft NoCs still trail hard NoCs in usable bandwidth, latency, area and routing congestion.

Recent Xilinx Versal and Achronix Speedster7t FPGAs integrate a hard NoC similar to the academic proposals discussed above. Versal uses a hard NoC for system-level communication between various endpoints (Gigabit transceivers, processor, AI subsystems, soft fabric), and is in fact the only way for external memory interfaces to communicate with the rest of the device (Swarbrick et al. 2019). It uses 128-bit wide links running at 1 GHz, matching a DDR channel's bandwidth. Its topology is related to a mesh, but with all horizontal links pushed to the top and bottom of the device to make it easier to lay out within the FPGA floorplan. The Versal NoC contains multiple rows (i.e., chains of links and routers) at the top and bottom of the device, and a number of vertical NoC columns (similar to any other hard block columns such as DSPs) depending on the device size as shown in Fig. 22a. The NoC has programmable routing tables that are configured at boot time and provides standard AXI interfaces as its fabric ports. The Speedster7t NoC topology is optimized for external interface to fabric transfers. It consists of a peripheral ring around the fabric with NoC rows and columns at regular intervals over the FPGA fabric as shown in Fig. 22b. The peripheral ring NoC can operate independently without configuring the FPGA fabric to route the traffic between different external interfaces. There is no direct connectivity between the NoC rows and columns; the packets from a master block connecting to a NoC row will pass through the peripheral ring to reach a slave block connected to a NoC column.

Interposers

FPGAs have been early adopters of interposer technology that allows dense interconnection of multiple silicon dice. As shown in Fig. 23a, a passive interposer is

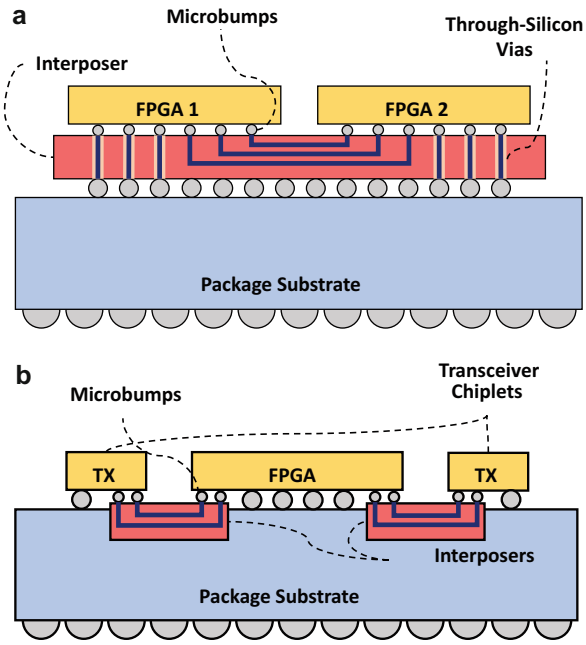

Fig. 23 Different interposer technologies used for integrating multiple chips in one package in: (**a**) Xilinx multi-die interposer-based FPGAs and (**b**) Intel devices with EMIB-connected transceiver chiplets

a silicon die (often in a trailing process technology to reduce cost) with conventional metal layers forming routing tracks and thousands of microbumps on its surface that connect to two or more dice flipped on top of it. One motivation for interposer-based FPGAs is achieving higher logic capacity at a reasonable cost. Both high-end systems and emulation platforms to validate ASIC designs before fabrication demand FPGAs with high logic capacity. However, large monolithic (i.e., single-silicon-die) devices have poor yield, especially early in the lifetime of a process technology (exactly when the FPGA is state-of-the-art). Combining multiple smaller dice on a silicon interposer is an alternative approach that can have higher yield. A second motivation for 2.5D systems is to enable integration of different specialized *chiplets* (possibly using different process technologies) into a single system. This approach is also attractive for FPGAs as the fabric's programmability can bridge disparate chiplet functionality and interface protocols.

Xilinx's largest devices starting from the Virtex-7 (28 nm) generation use passive silicon interposers to integrate three or four FPGA dice that each form a portion of the FPGA's rows. The largest interposer-based devices provide more than twice the logic elements of the largest monolithic FPGAs at the same process node. The FPGA programmable routing requires a large amount of interconnect, raising the question of whether the interposer microbumps (which are much larger and slower than conventional routing tracks) will limit the routability of the system. For example, in Virtex-7 interposer-based FPGAs, only 23% of the vertical routing tracks cross between dice through the interposer (Nasiri et al. 2015), with an estimated additional delay of ∼1 ns (Chaware et al. 2012). The study in Nasiri et al.

(2015) showed that CAD tools that place the FPGA logic to minimize crossing of an interposer boundary combined with architecture changes that increase the switch flexibility to the interposer-crossing tracks can largely mitigate the impact of this reduced signal count. The entire vertical bandwidth of the NoC in the Xilinx Versal architecture (discussed in the "System-Level Interconnect: Network-on-Chip" section) crosses between dice, helping to provide more interconnect bandwidth. An embedded NoC makes good use of the limited number of wires that can cross an interposer, as it runs its links at a high frequency and they can be shared by different communication streams as they are packet-switched. Xilinx has also used their interposer technology for heterogeneous integration by incorporating HBM, starting with their 16 nm Virtex Ultrascale+ generation.

Intel FPGAs instead use smaller interposers called embedded multi-die interconnect bridges (EMIB) carved into the package substrate as shown in Fig. 23b. Intel Stratix 10 devices use EMIB to integrate a large FPGA fabric die with smaller IO transceiver or HBM chiplets in the same package, decoupling the design and process technology choices of these two crucial elements of an FPGA. Some recent studies (Nurvitadhi et al. 2018, 2019) used EMIB technology to tightly couple an FPGA fabric with specialized ASIC accelerator chiplets for DL applications. This approach offloads specific kernels of the computation (e.g., matrix-matrix or matrix-vector multiplications) to the more efficient specialized chiplets, while leveraging the FPGA fabric to interface to the outside world and to implement rapidly changing DL model components.

Configuration and Security

An FPGA's *configuration circuitry* loads the bitstream into the millions of SRAM cells that control the LUTs, routing switches and configuration bits in hard blocks. On power up, a configuration controller loads this bitstream serially from a source such as on-board flash. When a sufficient group of configuration bits are buffered, they are written in parallel to a group of configuration SRAM cells, in a manner similar to writing a (very wide) word to an SRAM array. This configuration circuitry can also be accessed by the FPGA fabric and embedded processor subsystems, allowing *partial reconfiguration* of one part of the device while another portion continues processing. For high-reliability applications, this configuration circuitry can also be used to continuously read back the programmed configuration of the device and compute a cyclic redundancy check (CRC) in order to detect if any configuration SRAM cells have been upset by soft errors (such as those induced by high energy radiation).

A complete FPGA application is very valuable intellectual property, and without *security measures* it could be cloned simply by copying the programming bitstream. To avoid this, FPGA CAD tools can optionally encrypt a bitstream, and FPGA devices can have a private decryption key programmed in by the manufacturer to be used by the configuration controller, making a bitstream usable only by a single customer who purchases FPGAs with the proper key.

Conclusion

FPGAs have evolved from simple arrays of programmable logic blocks and IOs interconnected via programmable routing into complex multi-die systems with many different embedded components such as BRAMs, DSPs, high-speed external interfaces, and system-level NoCs. The recent adoption of FPGAs in the HPC and datacenter domains, along with the emergence of new high-demand applications such as deep learning, is ushering in a new phase of FPGA architecture design. These new applications and the multi-user paradigm of the datacenter create opportunities for architectural innovation. At the same time, process technology scaling is changing in fundamental ways. Wire delay is scaling poorly which motivates rethinking programmable routing architecture. Interposers and 3D integration enable entirely new types of heterogeneous systems. Controlling power consumption is an overriding concern, and is likely to lead to FPGAs with more power-gating and more heterogeneous hard blocks. We do not claim to predict the future of FPGA architecture, except that it will be interesting and different from today!

References

Abdelfattah MS, Betz V (2013) The case for embedded networks on chip on field-programmable gate arrays. IEEE Micro 34(1):80–89

Abdelfattah MS et al (2015) Take the highway: design for embedded NoCs on FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 98–107

Ahmed E, Rose J (2004) The effect of LUT and cluster size on deep-submicron FPGA performance and density. IEEE Trans Very Large Scale Integr (VLSI) Syst 12(3):288–298

Ahmed I et al (2019) FRoC 2.0: automatic BRAM and logic testing to enable dynamic voltage scaling for FPGA applications. ACM Trans Reconfig Technol Syst (TRETS) 12(4):1–28

Betz V, Rose J (1998) How much logic should go in an FPGA logic block? IEEE Des Test Comput 15(1):10–15

Betz V, Rose J (1999) FPGA routing architecture: segmentation and buffering to optimize speed and density. In: ACM International Symposium on FPGAs, pp 59–68

Betz V et al (1999) Architecture and CAD for deep-submicron FPGAs. Springer Science & Business Media. New York, USA

Bohr MT (1995) Interconnect scaling – the real limiter to high performance ULSI. In: Proceedings of International Electron Devices Meeting. IEEE, pp 241–244

Boutros A et al(2018) You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference. ACM Trans Reconfig Technol Syst (TRETS) 11(3):1–23

Boutros A et al (2018) Embracing diversity: enhanced DSP blocks for low-precision deep learning on FPGAs. In: IEEE International Conference on Field Programmable Logic and Applications (FPL), pp 35–357

Boutros A et al (2020) Beyond peak performance: comparing the real performance of AI-optimized FPGAs and GPUs. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 10–19

Boutros A et al (2022) Architecture and application co-design for beyond-FPGA reconfigurable acceleration devices. IEEE Access 10:95067–95082

Caulfield AM et al (2016) A cloud-scale acceleration architecture. In: IEEE/ACM International Symposium on Microarchitecture (MICRO), pp 1–13

Chaware R et al (2012) Assembly and reliability challenges in 3D integration of 28 nm FPGA die on a large high density 65 nm passive interposer. In: IEEE Electronic Components and Technology Conference, pp 279–283

Cheah HY et al (2014) The iDEA DSP block-based soft processor for FPGAs. ACM Trans Reconfig Technol Syst (TRETS) 7(3):1–23

Chiasson C, Betz V (2013a) COFFE: fully-automated transistor sizing for FPGAs. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 34–41

Chiasson C, Betz V (2013b) Should FPGAs abandon the pass gate? In: International Conference on Field-Programmable Logic and Applications, pp 1–8

Chromczak J et al (2020) Architectural enhancements in intel agilex FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 140–149

Ebeling C et al (2016) Stratix 10 high performance routable clock networks In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 64–73

Eldafrawy M et al (2020) FPGA logic block architectures for efficient deep learning inference. ACM Trans Reconfig Technol Syst (TRETS) 13(3):1–34

Estrin G (1960) Organization of computer systems: the fixed plus variable structure computer. In: Western Joint IRE-AIEE-ACM Computer Conference, pp 33–40

Feng W et al (2018) Improving FPGA performance with a S44 LUT structure. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 61–66

Fowers J et al (2018) A configurable cloud-scale DNN processor for real-time AI. In: ACM/IEEE International Symposium on Computer Architecture (ISCA), pp 1–14

Gaide B et al (2019) Xilinx adaptive compute acceleration platform: versal architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 84–93

Ganusov I, Devlin B (2016) Time-borrowing platform in the Xilinx ultrascale+ family of FPGAs and MPSoCs. In: IEEE International Conference on Field Programmable Logic and Applications (FPL), pp 1–9

Halfhill TR (2010) Tabula's time machine. Microprocess Rep 131:0–0

Hall M, Betz V (2020) From tensorflow graphs to luts and wires: automated sparse and physically aware CNN hardware generation. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 56–65

Hutton M et al (2005) Efficient static timing analysis and applications using edge masks. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 174–183

Kapre N, Gray J (2017) Hoplite: a deflection-routed directional torus NoC for FPGAs. ACM Trans Reconfig Technol Syst (TRETS) 10(2):1–24

Karandikar S et al (2018) FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In: International Symposium on Computer Architecture (ISCA). . IEEE, pp 29–42

Krupnova H, Saucier G (2000) FPGA-based emulation: industrial and custom prototyping solutions. In: International Workshop on Field-Programmable Logic and Applications (FPL). . Springer, pp 68–77

Kuon I, Rose J (2007) Measuring the gap between FPGAs and ASICs. IEEE Trans Comput-Aided Des Integr Circuit Syst 26(2):203–215

LaForest CE et al (2012) Multi-ported memories for FPGAs via XOR. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 209–218

Lai B-CC, Lin J-L (2016) Efficient designs of multiported memory on FPGA. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(1):139–150

Langhammer M, Pasca B (2015) Floating-point DSP block architecture for FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 117–125

Langhammer M et al (2021) Stratix 10 NX architecture and applications. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 57–67
Lemieux G et al (2000) Generating highly-routable sparse crossbars for PLDs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 155–164
Lemieux G et al (2004) Directional and single-driver wires in FPGA interconnect. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 41–48
Lewis D et al (2003) The Stratix routing and logic architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 12–20
Lewis D et al (2005) The Stratix II logic and routing architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 14–20
Lewis D et al (2009) Architectural enhancements in Stratix-III and Stratix-IV. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 33–42
Lewis D et al (2013) Architectural enhancements in Stratix V. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 147–156
Lewis D et al (2016) The Stratix 10 highly pipelined FPGA architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 159–168
Lockwood JW et al (2012) A low-latency library in FPGA hardware for high-frequency trading. In: Annual Symposium on High-Performance Interconnects (HOTI), pp 9–16
Meher PK et al (2008) FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic. IEEE Trans Signal Process 56(7):3009–3017
Murray K et al (2013) Titan: enabling large and complex benchmarks in academic CAD. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 1–8
Murray K et al (2020a) VTR 8: high-performance cad and customizable FPGA architecture modelling. ACM Trans Reconfig Technol Syst (TRETS) 13(2):1–55
Murray K et al (2020b) Optimizing FPGA logic block architectures for arithmetic. IEEE Trans Very Large Scale Integr (VLSI) Syst 28(6):1378–1391
Nasiri E et al (2015) Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs. IEEE Trans Very Large Scale Integr (VLSI) Syst 24(5):1821–1834
Nikolić S et al (2020) Straight to the point: intra- and intercluster LUT connections to mitigate the delay of programmable routing. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 150–160
Nurvitadhi E et al (2018) In-package domain-specific ASICs for intel Stratix 10 FPGAs: a case study of accelerating deep learning using TensorTile ASIC. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 106–1064
Nurvitadhi E et al (2019) Why compete when you can work together: FPGA-ASIC integration for persistent RNNs. In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 199–207
Papamichael MK, Hoe JC (2012) CONNECT: re-examining conventional wisdom for designing NoCs in the context of FPGAs. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 37–46
Parandeh-Afshar H et al (2012) Rethinking FPGAs: elude the flexibility excess of LUTs with and-inverter cones. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 119–128
Petelin O, Betz V (2016) The speed of diversity: exploring complex FPGA routing toplogies for the global metal layer. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 1–10
Petersen MB et al (2021) NetCracker: a peek into the routing architecture of Xilinx 7-series FPGAs. In: International Symposium on Field-Programmable Gate Arrays (FPGA)
Putnam A et al (2014) A reconfigurable fabric for accelerating large-scale datacenter services. In: ACM/IEEE International Symposium on Computer Architecture (ISCA), pp 13–24
Qian T et al (2018) A 1.25 Gbps programmable FPGA I/O buffer with multi-standard support. In: IEEE International Conference on Integrated Circuits and Microsystems, pp 362–365
Rasoulinezhad S et al (2019) PIR-DSP: an FPGA DSP block architecture for multi-precision deep neural networks. In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp 35–44

Rasoulinezhad S et al (2020) LUXOR: an FPGA logic cell architecture for efficient compressor tree implementations. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 161–171

Rettkowski J et al (2017) HW/SW co-design of the HOG algorithm on a xilinx zynq SoC. J Parallel Distrib Comput 109:50–62

Ronak B, Fahmy SA (2015a) Mapping for maximum performance on FPGA DSP blocks. IEEE Trans Comput-Aided Design Integr Circuits Syst 35(4):573–585

Ronak B, Fahmy SA (2015b) Minimizing DSP block usage through multi-pumping. In: International Conference on Field Programmable Technology (FPT)

Sivaswamy S et al (2005) HARP: hard-wired routing pattern FPGAs. In: International Symposium on Field-Programmable Gate Arrays (FPGA)

Swarbrick I et al Network-on-chip programmable platform in versal ACAP architecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 212–221

Tang X et al (2019) A study on switch block patterns for tileable FPGA routing architectures. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 247–250

Tatsumura K et al (2016) High density, low energy, magnetic tunnel junction based block RAMs for memory-rich FPGAs. In: IEEE International Conference on Field-Programmable Technology (FPT), pp 4–11

Tessier R et al (2007) Power-efficient RAM mapping algorithms for FPGA embedded memory blocks. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(2):278–290

Turakhia Y et al (2018) Darwin: a genomics co-processor provides up to 15,000x acceleration on long read assembly. ACM SIGPLAN Not 53(2):199–213

Tyhach J et al (2004) A 90 nm FPGA I/O buffer design with 1.6 Gbps data rate for source-synchronous system and 300 MHz clock rate for external memory interface. In: IEEE Custom Integrated Circuits Conference, pp 431–434

Upadhyaya P et al (2016) A fully-adaptive wideband 0.5–32.75 Gb/s FPGA transceiver in 16 nm FinFET CMOS technology. In: IEEE Symposium on VLSI Circuits, pp 1–2

Wang E et al (2019) Deep neural network approximation for custom hardware: where we've been, where we're going. ACM Comput Surv (CSUR) 52(2):1–39

Wilton S et al (1995) Architecture of centralized field-configurable memory. In: ACM International Symposium on Field-Programmable Gate Arrays (FPGA), pp 97–103

Wong H et al (2011) Comparing FPGA vs. custom cmos and the impact on processor microarchitecture. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 5–14

Yazdanshenas S, Betz V (2018) Interconnect solutions for virtualized field-programmable gate arrays. IEEE Access 6:10497–10507

Yazdanshenas S, Betz v (2019) COFFE 2: automatic modelling and optimization of complex and heterogeneous FPGA Architectures. ACM Trans Reconfig Technol Syst (TRETS), 12(1):1–27

Yazdanshenas S et al (2017) Don't forget the memory: automatic block RAM modelling, optimization, and architecture exploration. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 115–124

Yiannacouras P et al (2009) Data parallel FPGA workloads: software versus hardware. In: IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp 51–58

Young-Schultz T et al (2020) Using openCL to enable software-like development of an FPGA-accelerated biophotonic cancer treatment simulator. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 86–96

Zgheib G et al (2014) Revisiting and-inverter cones. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp 45–54

Zhao Z et al (2020) Achieving 100 Gbps intrusion prevention on a single server. In: USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp 1083–1100