FPGA Architecture: Principles and Progression

Andrew Boutros and Vaughn Betz

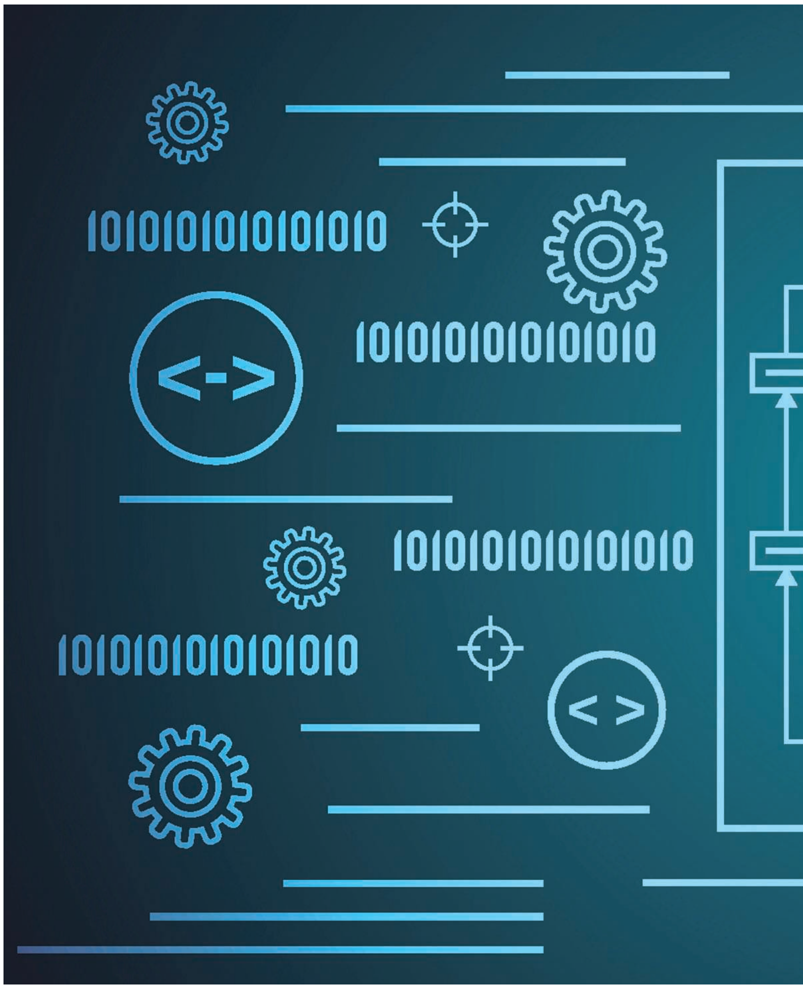
Abstract

Since their inception more than thirty years ago, field-programmable gate arrays (FPGAs) have been widely used to implement a myriad of applications from different domains. As a result of their low-level hardware reconfigurability, FPGAs have much faster design cycles and lower development costs compared to custom-designed chips. The design of an FPGA architecture involves many different design choices starting from the high-level architectural parameters down to the transistor-level implementation details, with the goal of making a highly programmable device while minimizing the area and performance cost of reconfigurability. As the needs of applications and the capabilities of process technology are constantly evolving, FPGA architecture must also adapt. In this article, we review the evolution of the different key components of modern commercial FPGA architectures and shed the light on their main design principles and implementation challenges.

I. Introduction

Field-programmable gate arrays (FPGAs) are reconfigurable computer chips that can be programmed to implement any digital hardware circuit. As depicted in Fig. 1, FPGAs consist of an array of different types of programmable blocks (logic, IO, and others) that can be flexibly interconnected using pre-fabricated routing tracks with programmable switches between them. The functionality of all the FPGA blocks and the configuration of the routing switches are controlled using millions of static random access memory (SRAM) cells that are programmed (i.e. written) at runtime to realize a specific function. The user describes the desired functionality in a hardware description language (HDL) such as Verilog or VHDL, or possibly uses high-level synthesis to translate C or OpenCL to HDL. The HDL design is then compiled using a complex computer-aided design (CAD) flow into the *bitstream* file used to program all the FPGA's configuration SRAM cells.

Digital Object Identifier 10.1109/MCAS.2021.3071607
Date of current version: 24 May 2021

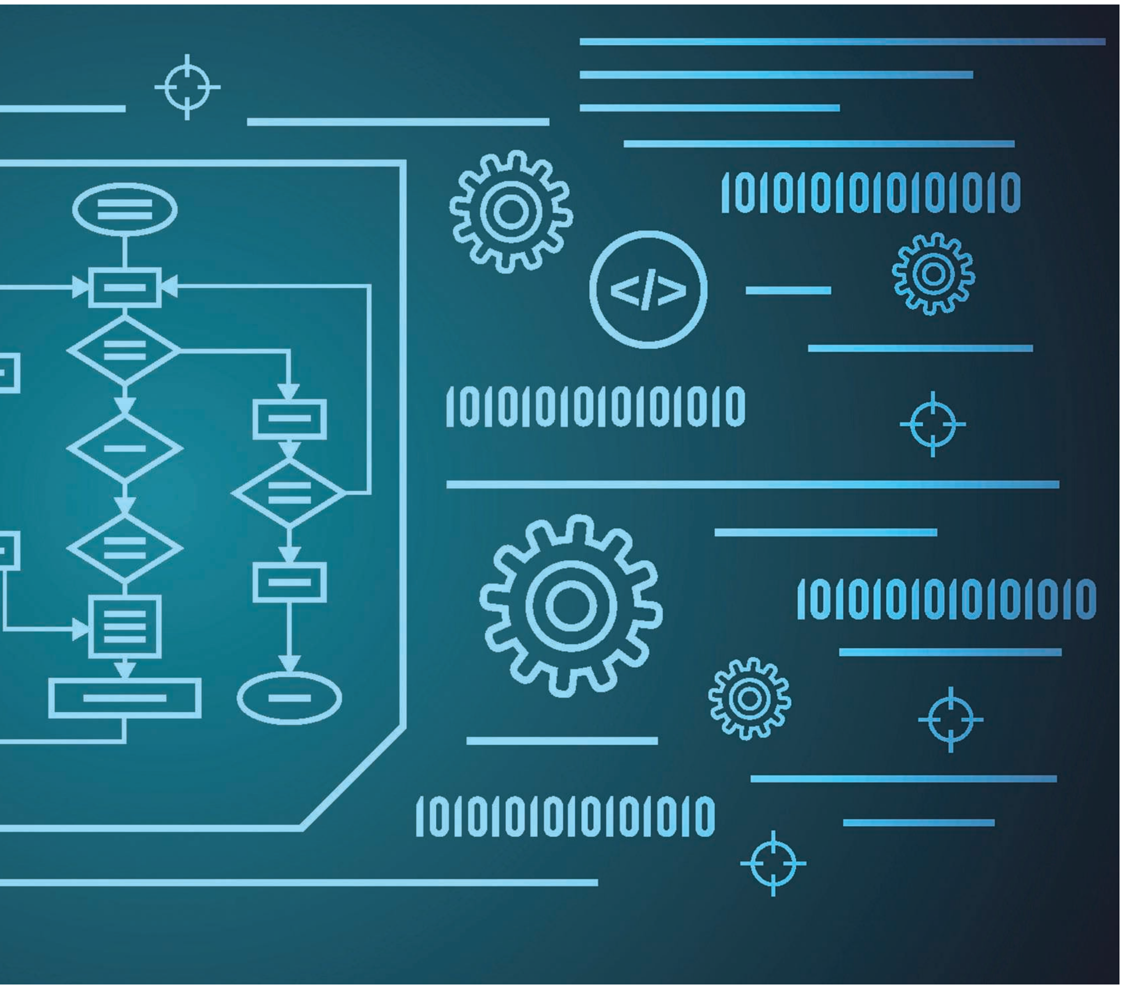


Compared to building a custom application-specific integrated circuit (ASIC), FPGAs have a much lower non-recurring engineering cost and shorter time-to-market. A pre-fabricated off-the-shelf FPGA can be used to implement a complete system in a matter of weeks, skipping the physical design, layout, fabrication, and verification stages that a custom ASIC would normally go through. They also allow 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. The bit-level reconfigurability of FPGAs enables implementation of 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 [1] or a processor *overlay* with an application-customized pipeline and instruction set [2].

These advantages motivated the adoption of FPGAs in many application domains including wireless communications, embedded signal processing, networking, ASIC prototyping, high-frequency trading, and many more [3]–[7]. They have also been recently deployed on a large scale in datacenters to accelerate search engines [8], packet processing [9], and machine learning [10] workloads, among others. However, the flexibility of FPGA hardware comes with an efficiency cost vs. ASICs. Kuon and Rose [11] 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 [12] shows that for full-featured designs which heavily utilize the other FPGA blocks (e.g. RAMs and DSPs), this area gap is reduced but is still 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.

In this article, we introduce key principles of FPGA architecture, and highlight the progression of these devices over the past 30 years. Fig. 1 shows how FPGAs evolved from simple arrays of programmable logic and IO blocks to complex heterogeneous multi-die systems with embedded block RAMs, digital signal processing (DSP) blocks, processor subsystems, diverse high-performance external interfaces, system-level interconnect, and more. First, we give a brief overview of the CAD flows and methodology used to evaluate new FPGA architecture ideas. We then detail the architecture challenges and design principles for each of the key components of an FPGA. We highlight key innovations in the design and implementation of each of these components over the past three decades along with areas of ongoing research.



©SHUTTERSTOCK.COM/ULVUR

II. FPGA Architecture Evaluation

As shown in Fig. 2, the FPGA architecture evaluation flow consists of three main components: a suite of benchmark applications, an architecture model, and a CAD system. Unlike an ASIC built for a specific functionality, an FPGA is a general-purpose platform designed for many use

Andrew Boutros and Vaughn Betz are with the Department of Electrical and Computer Engineering, University of Toronto and the Vector Institute for Artificial Intelligence. (email: andrew.boutros@mail.utoronto.ca, vaughn@eecg.utoronto.ca).

cases, some of which may not even exist when the FPGA is architected. Therefore, an FPGA architecture is evaluated based on its efficiency when implementing a wide 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 various customer applications. There are also several open-source benchmark suites such as the classic MCNC20 [13], the VTR [14], and the Titan23 [15] 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 use cases. The VTR and particularly the Titan suite are larger and more complex, making them more representative, but as FPGA capacity and application complexity continues to grow new benchmark suites are regularly needed.

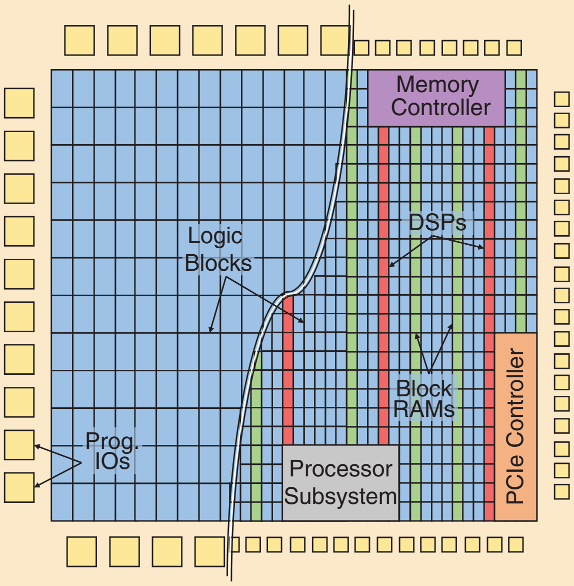


Figure 1. Early FPGA architecture with programmable logic and IOs vs. modern heterogeneous FPGA architecture with RAMs, DSPs, and other hard blocks. All blocks are interconnected using bit-level programmable routing.

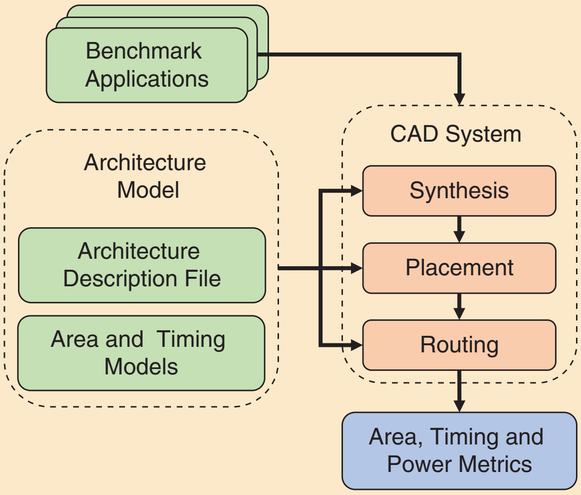


Figure 2. FPGA architecture evaluation flow.

The second part of the evaluation flow is the FPGA architecture model. The design of an FPGA involves many different decisions from architecture-level organization (e.g. number and type of blocks, distribution of wire segment lengths, size of logic clusters and logic elements) 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 (RAM, IO) even include significant analog circuitry. All these different components need to be carefully modeled to evaluate the FPGA architecture in its entirety. This is typically captured using an architecture description file that specifies the organization and types of the different FPGA blocks and the routing architecture, in addition to area, timing and power models obtained from circuit-level implementations for each of these components.

Finally, a re-targetable CAD system such as VTR [14] is used to map the selected benchmark applications on the specified 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 clock(s). 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 whether or not the

CAD tools fail to route some benchmarks on an 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 when implemented solely in the programmable fabric. 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 in the right spots to make this new hard block a net win.

In the following section, 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.

III. FPGA Architecture Evolution

A. Programmable Logic

The earliest reconfigurable computing devices were *programmable array logic* (PAL) architectures. PALs consisted of an array of `and` gates feeding another array of `or` gates, as shown in Fig. 3, and could implement any Boolean logic expression as a two-level sum-of-products function. PALs achieve configurability through programmable switches that 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 no matter what logic function is implemented. However, PALs do not scale well; as device logic capacity increased, the wires forming the `and`/`or` arrays became increasingly longer and slower and the number of programmable switches required 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 lookup-table-based (LUT-based) FPGA in 1984, which consisted of an array of SRAM-based 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

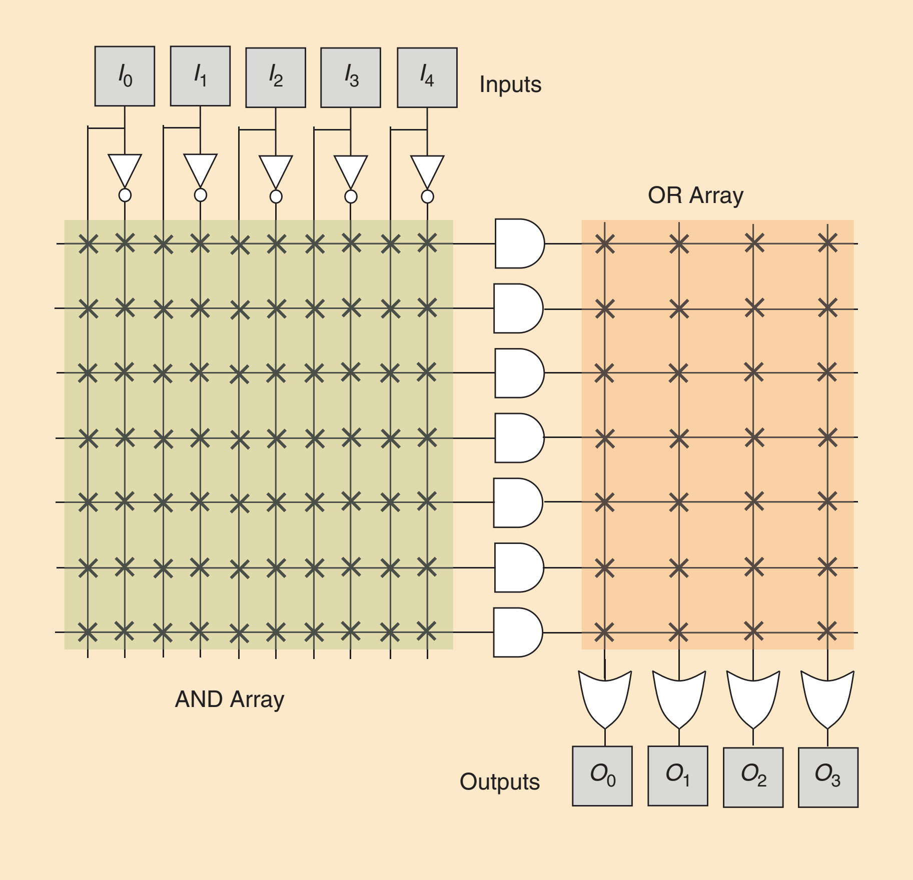


Figure 3. Programmable array logic (PAL) architecture with an `and` array feeding an `or` array. The crosses are reconfigurable switches that are used to program any Boolean expression as a two-level sum-of-products function.

element in all commercial FPGAs. Several research attempts [16]–[18] 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). However, when thoroughly evaluated in [19], AIC-based FPGA architectures had significantly larger area than LUT-based ones, with delay gains only on small benchmarks that have short critical paths.

A *K*-LUT can implement any *K*-input Boolean function by storing its truth table in configuration SRAM cells. *K* input signals are used as multiplexer select lines to choose an output from the 2^K values of the truth table. Fig. 4(a) shows the transistor-level circuit implementation

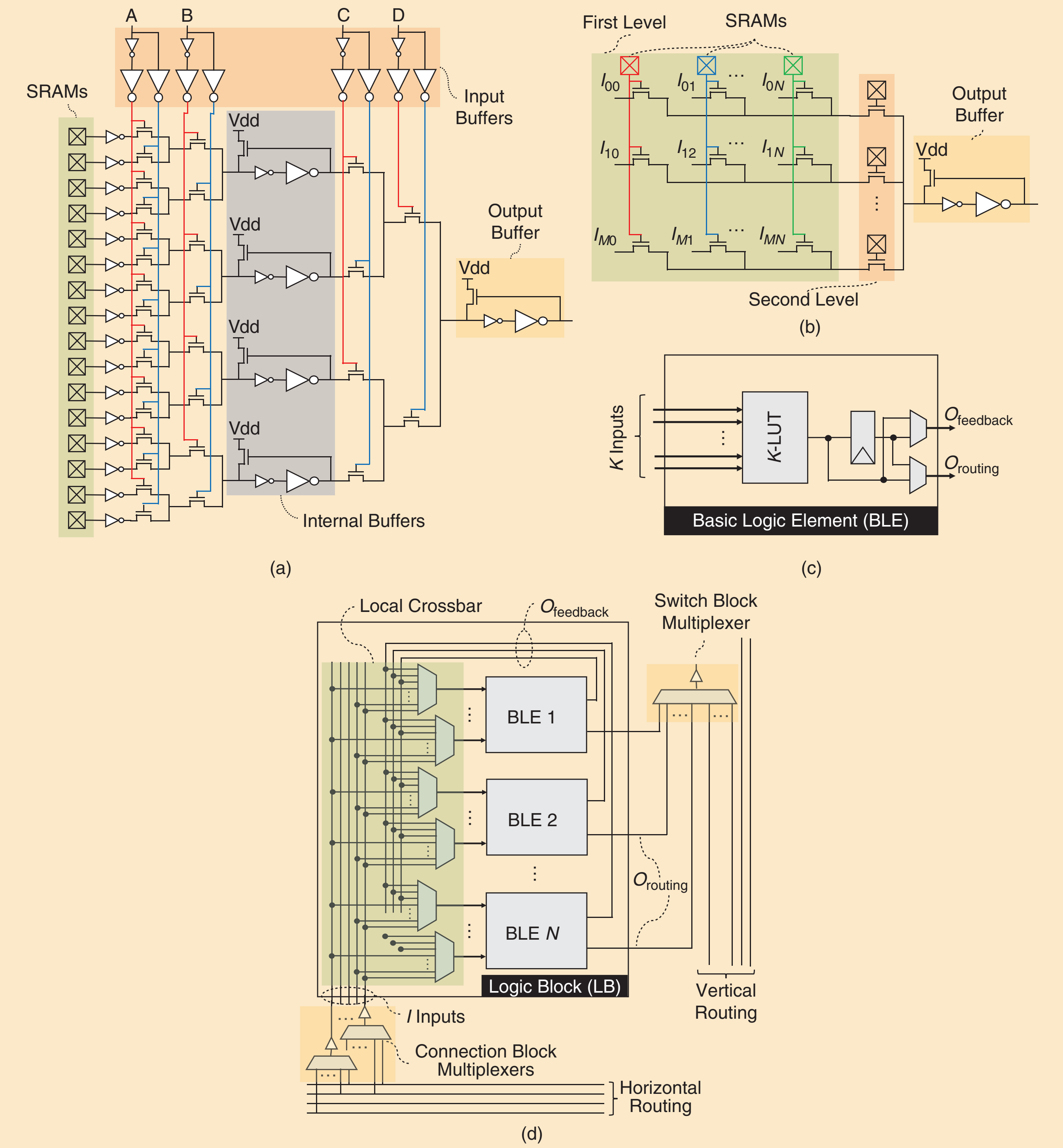


Figure 4. (a) Transistor-level implementation of a 4-LUT with internal buffers between the second and third LUT stages, (b) Two-level multiplexer circuitry, (c) Basic logic element (BLE), and (d) Logic block (LB) internal architecture.

of a 4-LUT using pass-transistor logic. In addition to the output buffer, an internal buffering stage (shown between the second and third stages of the LUT in Fig. 4(a)) 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 [20] defines the *basic logic element* (BLE) as a K-LUT coupled with an output register and bypassing 2:1 multiplexers as shown in Fig. 4(c). Thus, a BLE can either implement just a flip-flop (FF) or a K-LUT with registered or unregistered output. As illustrated in Fig. 4(d), 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 [21] or partial [22] 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. 4(b); this is the most efficient circuit design for FPGA multiplexers in most cases [23]. Fig. 4(d) also shows the switch and connection block multiplexers forming the programmable routing that allows logic blocks to connect to each other; this routing is discussed in detail in Section III-B.

Over the years, the size of both LUTs (K) and LBs (N) have gradually increased as device logic capacity has grown. As K increases, more functionality can be packed into a single LUT, reducing not only the number of LUTs needed but also the 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 (as the multiplexer constitutes a chain of K pass transistors with periodic buffering). If the LB local interconnect is implemented as a crossbar, its size increases quadratically and its speed degrades linearly with the number of BLEs in the LB, N. Ahmed and Rose [24] empirically evaluated these trade-offs and concluded 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 LUT-based FPGA from Xilinx, the XC2000 series in 1984, had an LB that contained only two 3-LUTs (i.e. $N = 2, K = 3$). LB size gradually increased over time and by 1999, Xilinx's Virtex family included four 4-LUTs and Altera's Apex 20K family included ten 4-LUTs in each LB.

The next major change in architecture came in 2003 from Altera, with the introduction of *fracturable* LUTs in their Stratix II architecture [25]. Ahmed and Rose in [24] showed that an LB with ten 6-LUTs achieved 14% higher performance than a LB with ten 4-LUTs, but at a 17% higher area. Fracturable LUTs seek to combine the best of both worlds, achieving the performance of a larger LUT with the area-efficiency of smaller LUTs. A major factor in the area increase with traditional 6-LUTs is under-utilization: Lewis et al. found that 64% of the LUTs in benchmark applications used fewer than 6 inputs, wasting some of a 6-LUT's functionality [26]. 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. Fig. 5(a) 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, requiring the two 5-LUTs to share all their

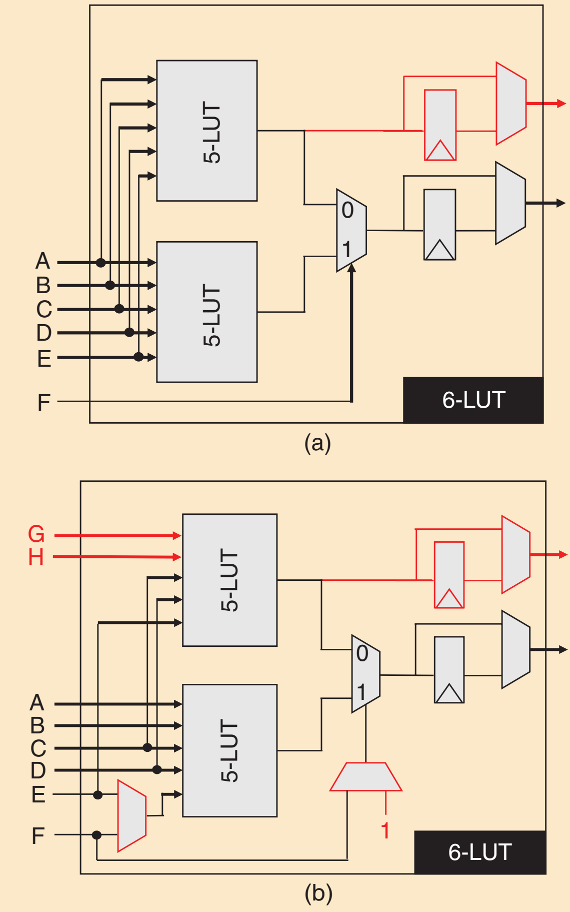
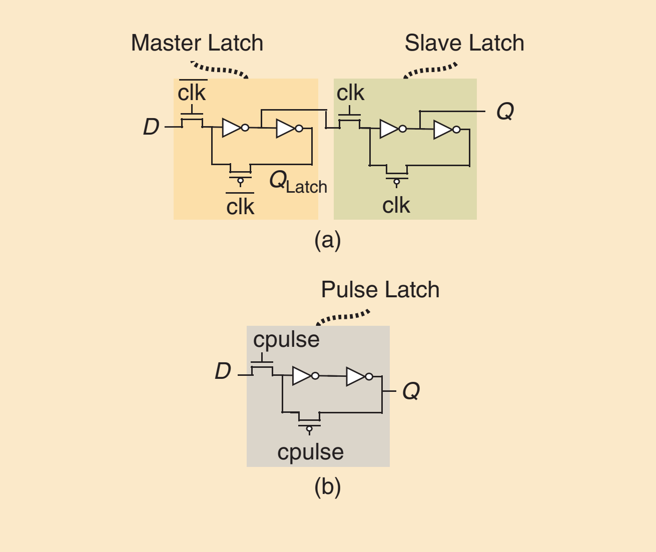


Figure 5. 6-LUT fracturable into two 5-LUTs with (a) no additional input ports, leading to 5 shared inputs (A-E) or (b) two additional input ports and steering multiplexers, leading to only 2 shared inputs (C, D).

inputs will limit how often both can be simultaneously used. Adding extra routing ports as shown in Fig. 5(b) increases the area of the fracturable 6-LUT, but makes it easier to find two logic functions that can be packed together into it. 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 performance improvement. On the other hand, 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 [26].

Xilinx later adopted a related fracturable LUT approach in their Virtex-5 architecture. Like Stratix II, a Virtex-5 6-LUT can be decomposed into two 5-LUTs. However, Xilinx chose to minimize the extra circuitry added for fracturability as shown in Fig. 5(a)—no extra input routing ports or steering multiplexers are added. 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 [27]. While subsequent architectures from both Altera/Intel and Xilinx have also been based on fracturable 6-LUTs, a recent Microsemi study [28] revisited the 4-LUT vs. 6-LUT efficiency trade-off for newer process technologies, CAD tools and designs than those used in [24]. It shows that a LUT structure with two tightly coupled 4-LUTs, one feeding the other, can achieve performance close to plain 6-LUTs along with the area and power advantages 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 recently announced Versal architecture from Xilinx further increases the number of BLEs per LB to thirty two [29]. The reasons for this large increase are two-fold. 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 help mitigate this trend by simplifying placement and inter-LB routing.



(a)

(b)

Figure 6. Circuitry for (a) Master-slave positive-edge-triggered flip-flop, and (b) Pulse latch.

Another important architecture choice is the number of FFs per BLE. Early FPGAs coupled a (non-fracturable) LUT with a single FF as shown in Fig. 4(c). 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. 5(a) and 5(b). In the Stratix V architecture, the number of FFs was further increased from two to four per BLE in order to accommodate increased demand for FFs as designs became more deeply pipelined to achieve higher performance [30]. 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. 6(b), this removes one of the two latches that would be present in a master-slave FF (Fig. 6(a)), 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. If a pulse generator was built for each FF, the overall area per FF would increase rather than decrease. Instead, Stratix V contains 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. Longer pulses further degrade hold time, but generally any hold violations can be solved by the FPGA routing algorithm using longer wiring paths to delay signals. Xilinx also uses pulse latches as its FFs in its Ultrascale+ architecture [31].

Arithmetic operations (add and subtract) are very common in FPGA designs: Murray et al. found that 22% of the logic elements in a suite of FPGA designs were implementing arithmetic [32]. While these operations can be implemented with LUTs, each bit of arithmetic in a ripple carry adder requires two LUTs (one for the sum output and one for the carry). This leads to both high logic utilization and a slow critical path due to connecting many LUTs in series to compute carries for multi-bit additions. Consequently, all modern FPGA architectures include hardened arithmetic circuitry in their logic blocks. 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 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 [32]). Hardening more sophisticated structures like carry skip adders further improves speed (an additional 20% speed-up at 32-bits in [33]). The latest Versal architecture from Xilinx 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. 7(a) [29]. This organization allows implementing 1-bit of arithmetic per logic element. On the other hand, the latest Intel Agilex architecture can implement 2-bits of arithmetic per logic element, with dedicated interconnect for the carry between logic elements as shown in Fig. 7(b). It achieves that by hardening 2-bit carry-skip adders that are fed by the four 4-LUTs contained within a 6-LUT [34]. The study by Murray et al. [32] 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 Section III-E; 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 [9]. LUTs are used to generate the partial products of a multiplier array followed by an adder tree to reduce the partial products and perform the accumulation. Consequently, multiple recent studies [35]–[37] 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 acceleration. The work in [36] and [37] 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 2 N inputs are needed to generate N^2 partial products). The most promising of these proposals increases the density of MAC operations by 1.7× 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.

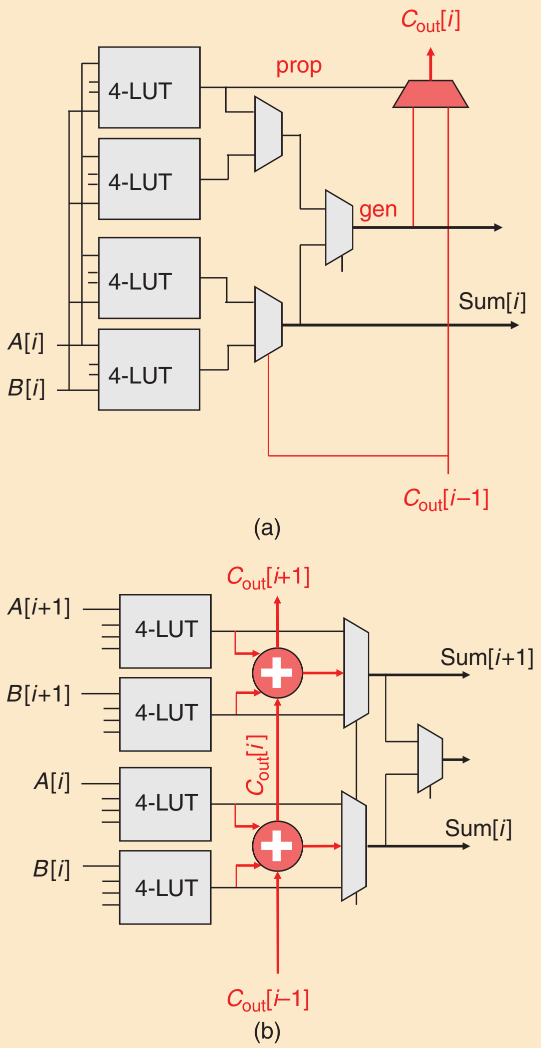


Figure 7. Overview of the hard arithmetic circuitry (in red) in the logic elements of (a) Xilinx and (b) Altera/Intel FPGAs. $A[i]$ and $B[i]$ are the ith bits of the two addition operands A and B. The Xilinx LEs compute carry propagate and generate in the LUTs, while the Altera/Intel ones use LUTs to pass inputs to the hard adders. Unlabled inputs are unused when implementing adders.

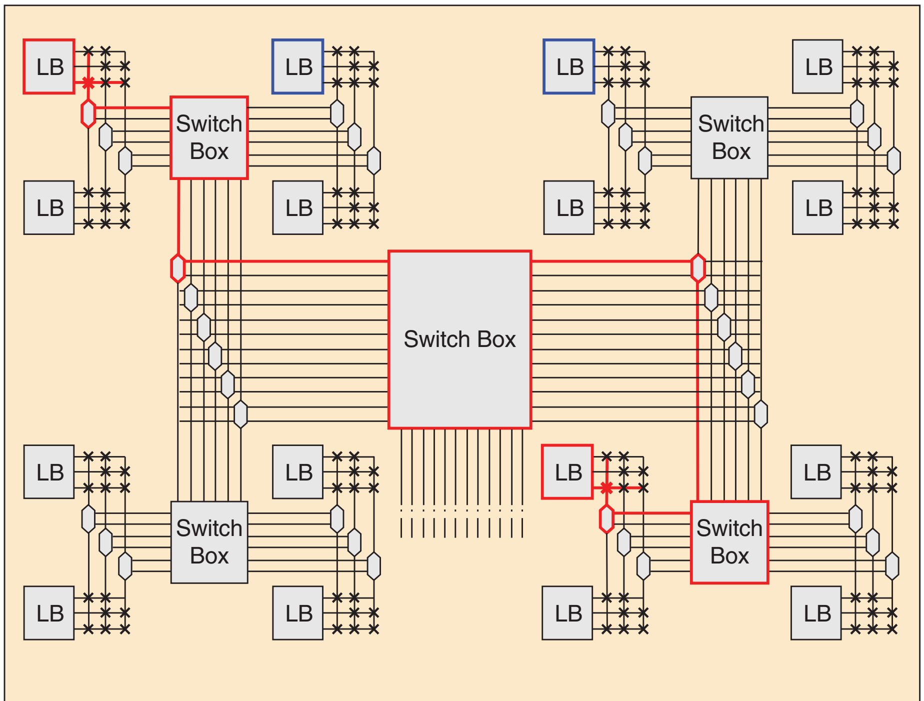


Figure 8. Routing architecture in hierarchical FPGAs.

B. Programmable Routing

Programmable routing commonly accounts for over 50% of both the fabric area and the critical path delay of applications [38], so its efficiency is crucial. Programmable routing is composed of pre-fabricated wiring segments and programmable switches. By programming an appropriate sequence of switches to be on, any function block output can be connected to any input. 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. Communication is more frequent between modules that are near each other in the design hierarchy, and hierarchical FPGAs can realize these connections with short wires that connect small regions of a chip. As shown in Fig. 8, to communicate to more distant regions of a hierarchical FPGA, 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 7K and Apex 20K 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 8) which still require several wires and switches to connect. Consequently it is primarily used today for smaller FPGAs, such as the FlexLogix FPGA IP cores that can be embedded in larger SoC designs [39].

The other type of FPGA interconnect is *island-style*, as depicted in Fig. 9. 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. 9, 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. 9.

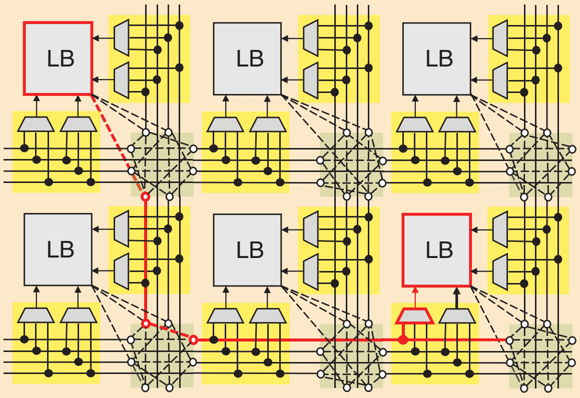


Figure 9. Island-style routing architecture. Thick solid lines are routing wires while dashed lines are programmable switches. Connection and switch blocks are shaded in yellow and green, respectively.

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. 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. Some of the routing architecture parameters 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 [20]. In Fig. 9 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 for target applications and at a specific process node requires experimentation using a full CAD flow as detailed in Section II.

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% [40]. 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 [41]. 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 [42]. 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. 4(d), most logic blocks now group multiple BLEs together with local routing. This means 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 end point. The local routing in a logic cluster (described in Section III-A) allows some block inputs and some block outputs to be swapped during routing. 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 [43].

There are also multiple options for the electrical design of programmable switches, as shown in Fig. 10. Early FPGAs used pass gate transistors controlled by SRAM cells to connect wires. While this is the smallest switch possible in a conventional CMOS process, the delay of routing wires connected in series by pass transistors grows quadratically, making them very slow for large FPGAs. Adding some tri-state buffer switches costs area, but improves speed [40]. 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. 4(b). 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 [42], [44]. 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 Section III-G. These wires often have multiple tri-state buffers that can drive them, as the cost of

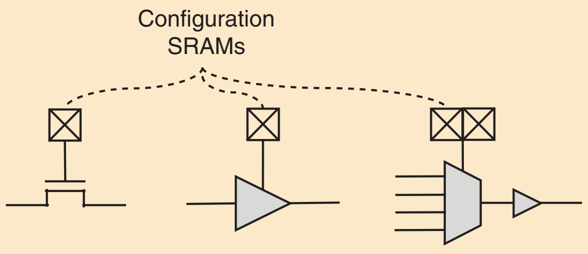


Figure 10. Different implementations for SRAM-controlled programmable switches using pass transistors (left), tri-state buffers (middle), or buffered multiplexers (right).

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 6(b), 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 [45]. Hold time concerns mean that using pulse latches in immediately consecutive Stratix 10 routing switches is not possible, so Intel refined this approach in their next-generation Agilex devices by integrating actual registers (with better hold time) on only one-third of the interconnect drivers (to mitigate the area cost) [34]. 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 [46].

C. Programmable IO

FPGAs include unique programmable IO structures to allow them to communicate with a very wide variety of other devices, making FPGAs the communications hub of many systems. For a single set of physical IOs to *programmably* support many different IO interfaces and standards is challenging, as 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's Stratix II (90 nm) devices support 28 different IO standards and devote 20% (largest device) to 48% (smallest device) of their die area to IO-related structures.

As Fig. 11 shows, FPGAs address this challenge using a combination of approaches [47]–[49]. First, FPGAs use IO buffers that can operate across a range of voltages. 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 buffer. This allows different banks to operate at different voltage levels; e.g. 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 form the positive and negative line for differential IO standards. 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. By programming some pull-up or pull-down transistors to be enabled even when no output is being driven, FPGA IOs can also be programmed to implement different on-chip termination resistances to minimize signal reflections. Programmable delay chains 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. 11 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 transfer to the fabric at a lower frequency. Most FPGAs now also contain by-passable higher-level 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 with less voltage and electrical programmability; they can only be used as serial transceivers [50]. 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, distributing the very high data bandwidths from IO interfaces requires wide soft buses in the fabric, which creates additional challenges as discussed later in Section III-F.

D. On-Chip Memory

The first form of on-chip memory elements in FPGA architectures was FFs integrated in the FPGA's logic blocks as described in Section III-A. However, as FPGA logic capacity grew, they were used to implement larger systems which almost always require memory to buffer and

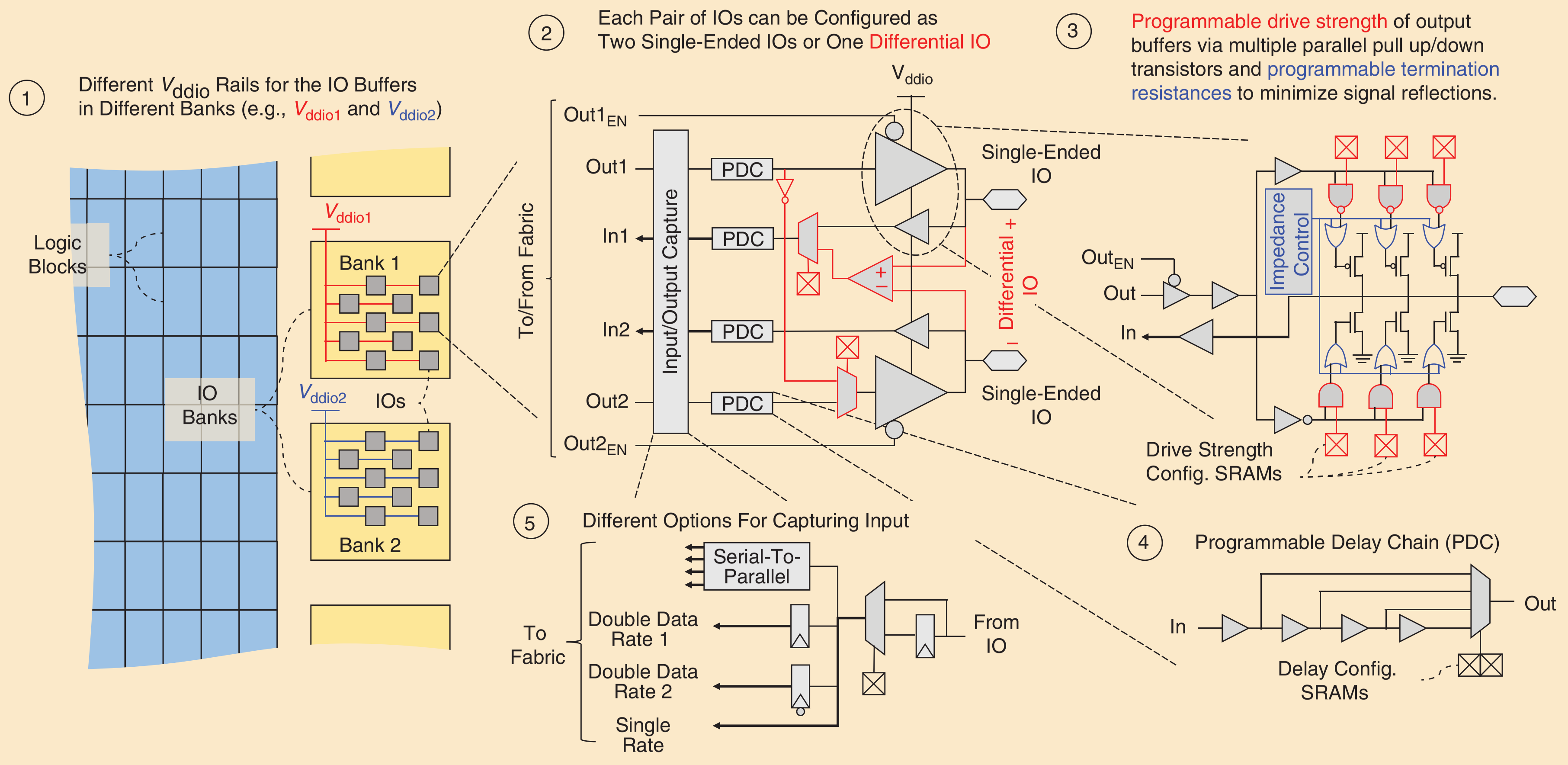

1
Different V_{ddio} Rails for the IO Buffers in Different Banks (e.g., V_{ddio1} and V_{ddio2})
Logic Blocks
IO Banks
V_{ddio1}
Bank 1
V_{ddio2}
IOs
Bank 2
2
Each Pair of IOs can be Configured as Two Single-Ended IOs or One Differential IO
V_{ddio}
To/From Fabric
$Out1_{EN}$
Out1
In1
In2
Out2
$Out2_{EN}$
Input/Output Capture
PDC
PDC
PDC
PDC
Single-Ended IO
Differential + − IO
Single-Ended IO
3
Programmable drive strength of output buffers via multiple parallel pull up/down transistors and programmable termination resistances to minimize signal reflections.
Out_{EN}
Out
In
Impedance Control
Drive Strength Config. SRAMs
4
Programmable Delay Chain (PDC)
In
Out
Delay Config. SRAMs
5
Different Options For Capturing Input
Serial-To-Parallel
To Fabric
Double Data Rate 1
Double Data Rate 2
Single Rate
From IO

Figure 11. Overview of the different techniques for implementing programmable IOs on FPGAs.

re-use data on chip, making it highly desirable to have denser on-chip storage since building large RAMs out of registers and LUTs is over 100× less dense than an (ASIC-style) SRAM block. At the same time, the RAM needs 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) used universally in 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 [51]. It included columns of small (2 kb) BRAMs that connect to the rest of the fabric through the programmable routing. FPGAs have gradually incorporated larger and more diverse BRAMs and it is typical for ~25% of the area of a modern FPGA to be dedicated for BRAMs [52].

An FPGA BRAM consists of an SRAM-based memory *core*, with additional *peripheral circuitry* to make them more configurable for multiple purposes and to connect them to the programmable routing. An SRAM-based BRAM is typically organized as illustrated in Fig. 12. It 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/

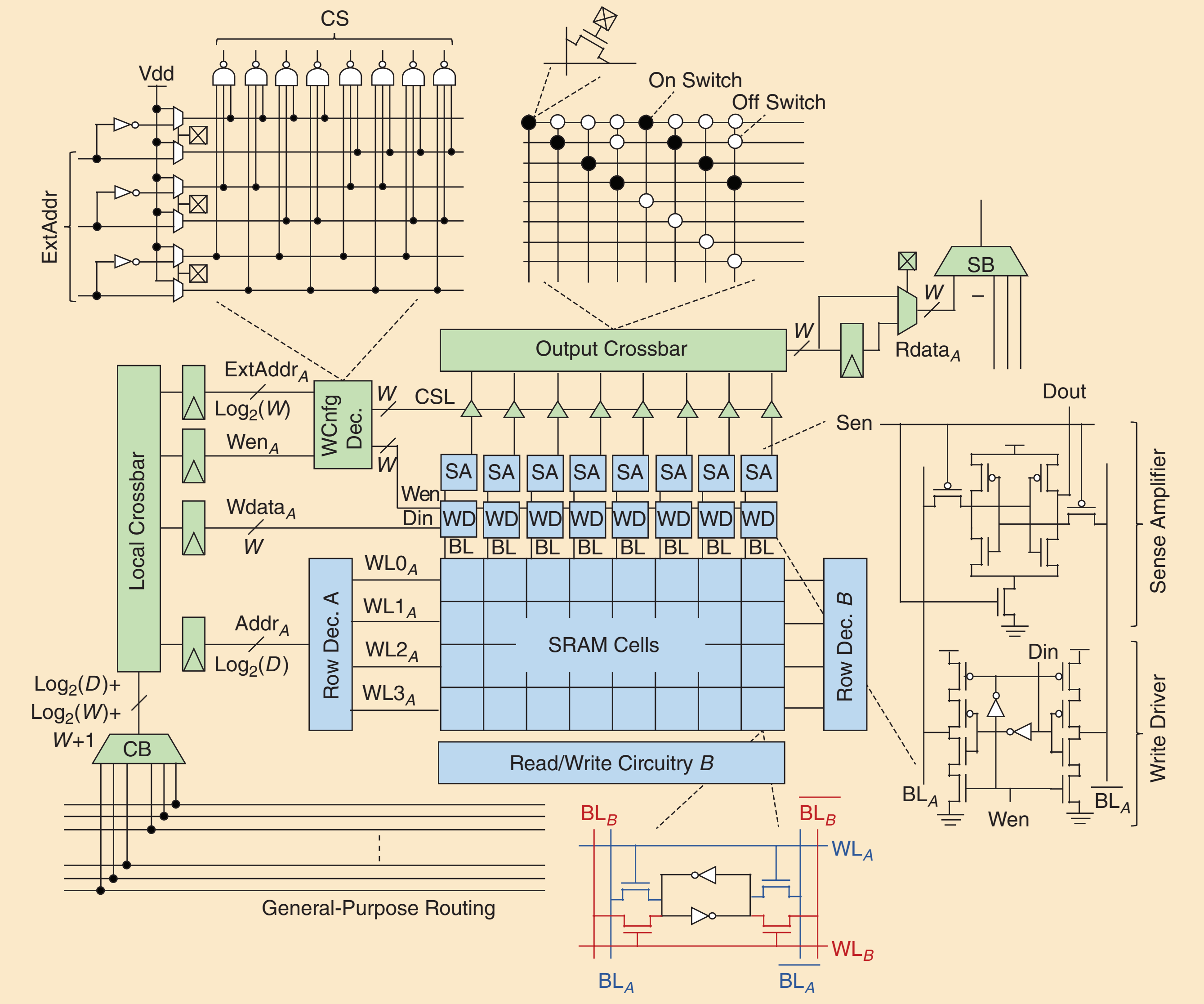


Figure 12. Organization and circuitry of a conventional dual-port SRAM-based FPGA BRAM. The components highlighted in blue are common in any SRAM-based memory module, while those highlighted in green are FPGA-specific. This BRAM has a maximum data width of 8 bits, but the output crossbar is configured for 4-bit output mode.

write operations. To simplify timing of the read and write operations, all modern FPGA BRAMs register all their inputs. During a write operation, the column decoder activates the write drivers, 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 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 circuit to produce the output data [52].

The main architectural decisions in designing FPGA BRAMs are choosing their capacity, data word width, and number of read/write ports. 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. The area occupied by the SRAM 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 wasted. Similarly, a BRAM with a 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. 12. 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. 12). 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 [53], [54]. 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. 12 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 width configurability decoder (WCnfg Dec.) selects between V_{dd} and address bits, as shown in the top-left of Fig. 12 for a maximum word size of 8 bits. 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 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 Section III-A; 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% to 35% of the BRAM tile area for 256Kb down to 8Kb BRAMs, respectively [55]. This motivates minimizing the number of routing ports to a BRAM as much as possible without unduly comprising its functionality. Table I 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

Table I.
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	$4\log_2(D) + 4W$

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 2 W 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 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 (i.e. operate 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 [56]. Multi-pumping can also be used with conventional FPGA BRAMs by building the time-multiplexing logic in the soft fabric; however, this leads to aggressive timing constraints for the time-multiplexing logic, which can make timing closure more challenging and increase compile time. 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 [57]. 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 [58], [59]. We refer the interested reader to both [52] and [55] for extensive details about the design of BRAM core and peripheral circuitry.

In addition to building BRAMs, FPGA vendors can add circuitry that allows designers to repurpose the LUTs that form the logic fabric into additional RAM blocks. The truth tables in the logic block K-LUTs are 2^K ×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 distributed LUT-based RAMs (LUT-RAMs) just by adding 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 Section III-A. 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 are assembled by bringing one signal in from an unused routing port in each ALM and broadcasting the resulting address and enable to all ALMs [60]. 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 (broadcast) 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 convert more than 50% of the logic fabric to LUT-RAMs, both Altera/Intel and Xilinx have elected to make only half 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 also would tie the design to a specific 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. Fig. 13 gives 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. 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 the most-significant bits of the write and read addresses are used as write enable and read output mux 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 (see Fig. 13) or double-pumped as previously explained. Several algorithms for optimizing RAM mapping are described in [61], [62].

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. Fig. 14 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 10 K 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, 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 9 kb and 144 kb BRAM. 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

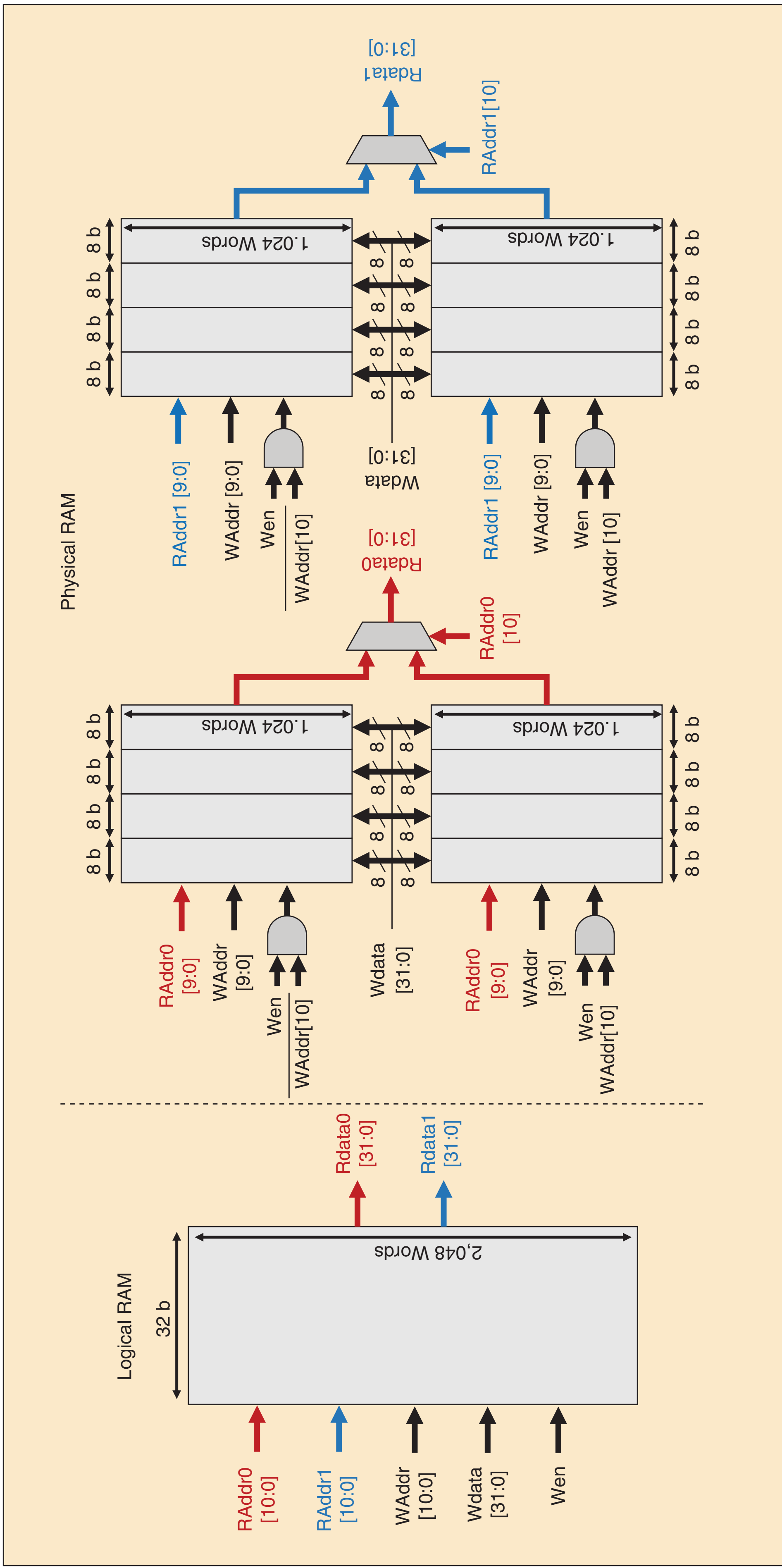


Figure 13. Mapping a 2048 × 32-bit 2r+1w logical RAM to an FPGA with 1024 × 8-bit 1r+1w physical BRAMs.

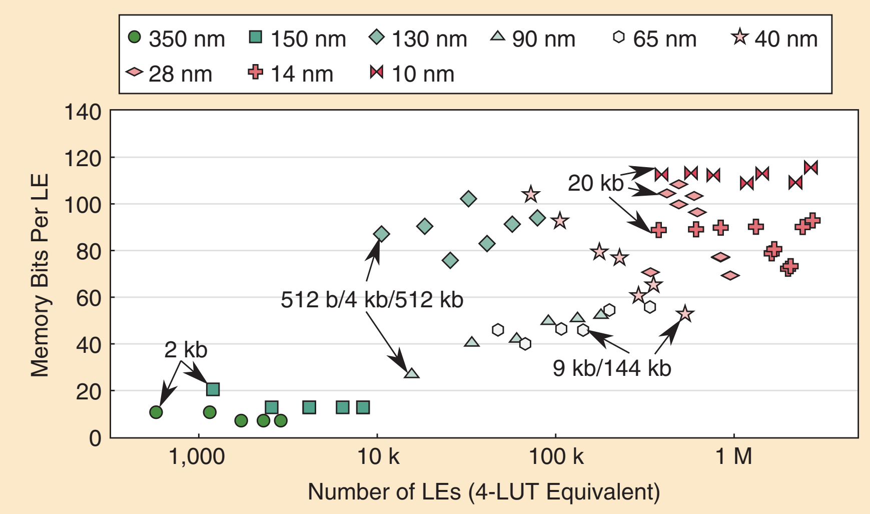


Figure 14. Trends in memory bits per LE for Altera/Intel FPGAs starting from the 350 nm Flex 10k (1995) to the 10 nm Agilex (2019) architecture. The labels show the sizes of BRAMs in each of these architectures.

simplify both the FPGA layout as well as RAM mapping and placement. Tatsumura et al. [52] plot a similar trend for on-chip memory density in Xilinx devices as well. Similarly to Intel, Xilinx's RAM architecture 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 have also included 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.

To give some insight into the relative areas and efficiencies of different RAM blocks, Table II 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 [63] 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 shows how important on-chip RAM is: area is over 300× larger than the 144 kb BRAM. 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 [55]. Published studies into BRAM architecture trade-offs for FPGAs include [30], [55], [64].

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-rich FPGAs and SRAM scaling becoming increasingly difficult due to process variation, a few academic studies (e.g. [52], [65]) have explored the use of other emerging memory technologies such as magnetic tunnel junctions (MTJs) to build FPGA memory blocks. According to [52], 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.

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

		BRAMs			
Implementation	**half-ALM**	**9k**	**144k**	**Area (mm^2)**	**Freq. (Mhz)**
144kb BRAMs	0	0	1	0.22 (1.0×)	336 (1.0×)
9kb 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×)

E. DSP Blocks

Initially the only dedicated arithmetic circuits in commercial FPGA architectures were carry chains to implement efficient adders, as discussed in Section III-A. Thus multipliers had to be implemented in the soft logic using LUTs and carry chains, incurring a substantial area and delay penalty. As high-multiplier-density signal processing and communication applications constituted a major FPGA market, 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 finite impulse response (FIR) filter structures on LUT-based FPGAs [66], [67].

With the prevalence of multipliers in FPGA designs from key application domains, and their lower area/delay/power 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 elements with only 2 N inputs and outputs. Therefore, the gains of hardening the multiplier logic and the cost of the programmable interfaces to the FPGA's routing fabric resulted in a net efficiency gain and strongly advocated for adopting hard multipliers in subsequent FPGA architectures. As shown at top left of Fig. 15, Xilinx introduced its Virtex-II architecture with the industry's first 18 × 18 bit hard multiplier blocks [68]. 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. Multiple hard 18-bit multipliers could be combined to form bigger multipliers or FIR filters using soft logic resources.

In 2002, Altera adopted a different approach by introducing full featured DSP blocks targeting the communications and signal processing domains in their Stratix architecture [42] (see second block in Fig. 15). 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 make use of it. The Stratix DSP block was highly configurable with support for different modes of operation and multiplication precisions unlike the fixed-function hard 18-bit multipliers in the Virtex-II architecture. Each Stratix variable-precision DSP block spanned 8 rows and could implement eight 9 × 9 bit multipliers, four 18 × 18 bit multipliers, or one 36 × 36 multiplier.

These modes of operation selected by Altera highlight an important theme of designing FPGA hard blocks: increasing the configurability and utility of these blocks by adding low-cost circuitry. 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. Fig. 16 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, the 18 × 18 array is split into only two 9 × 9 arrays (colored blue in Fig. 16). 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

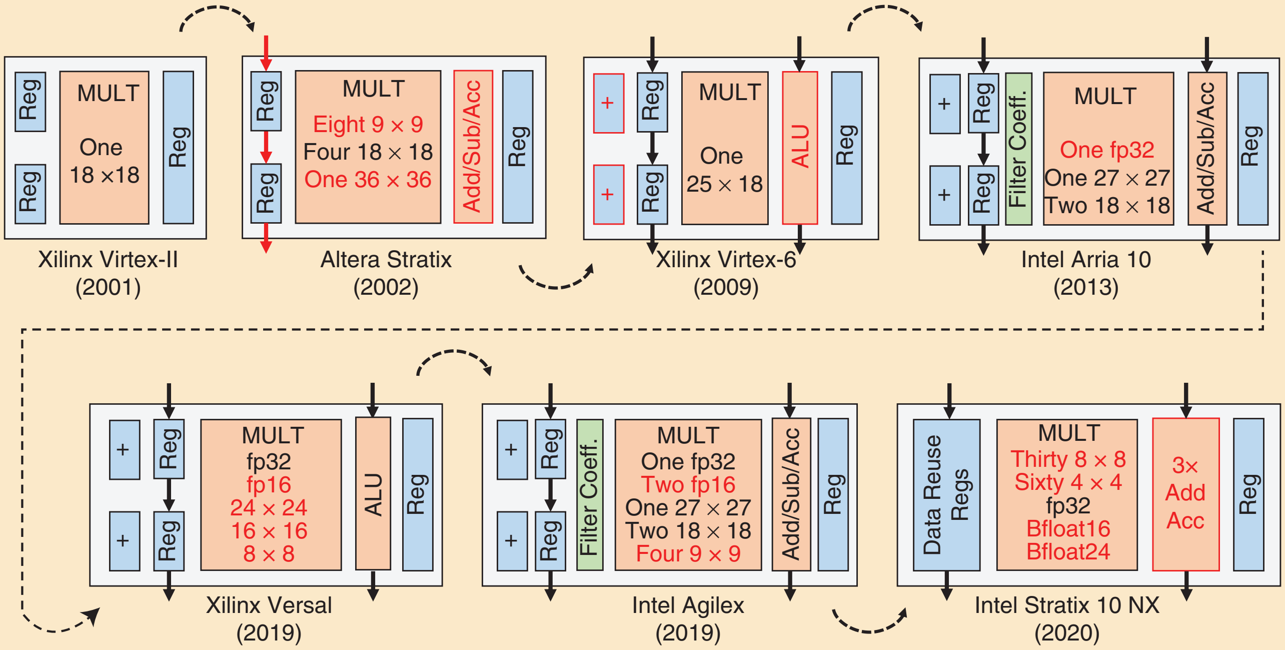


Figure 15. DSP block evolution in Altera/Intel and Xilinx FPGAs. Incrementally added features are highlighted in red.

array, marked with crosses in Fig. 16 to implement two's complement signed multiplication using the Baugh-Wooley algorithm [69] (the bottom left array already has the inverting capability from the 18 × 18 array).

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 [70]. The latest 28 nm architectures from Lattice also have a variable precision DSP block that can implement the same range of precisions, in addition to special 1 D and 2 D symmetry modes for filter structures and video processing applications, respectively. Xilinx also adopted a full-featured DSP block approach by introducing their *DSP48* tiles in the Virtex-4 architecture [71]. 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 to implement high-speed systolic FIR filters with hardened reduction chains.

An *N*-tap FIR filter performs a discrete 1 D 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).

$$y_n = c_0 x_n + c_1 x_{n-1} + \cdots + c_N x_{n-N} = \sum_{i=0}^{N} c_i x_{n-i} \qquad (1)$$

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).

$$y_n = c_0[x_n + x_{n-N}] + \cdots + c_{N/2-1}[x_{n-N/2-1} + x_{n-N/2}] \qquad (2)$$

Fig. 17 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 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 DSP block in the proper mode. Consequently, using the more powerful DSP block features makes a design less portable.

The Stratix III 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 [72]. Table III lists the implementation results of both symmetric and asymmetric 51-tap 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.

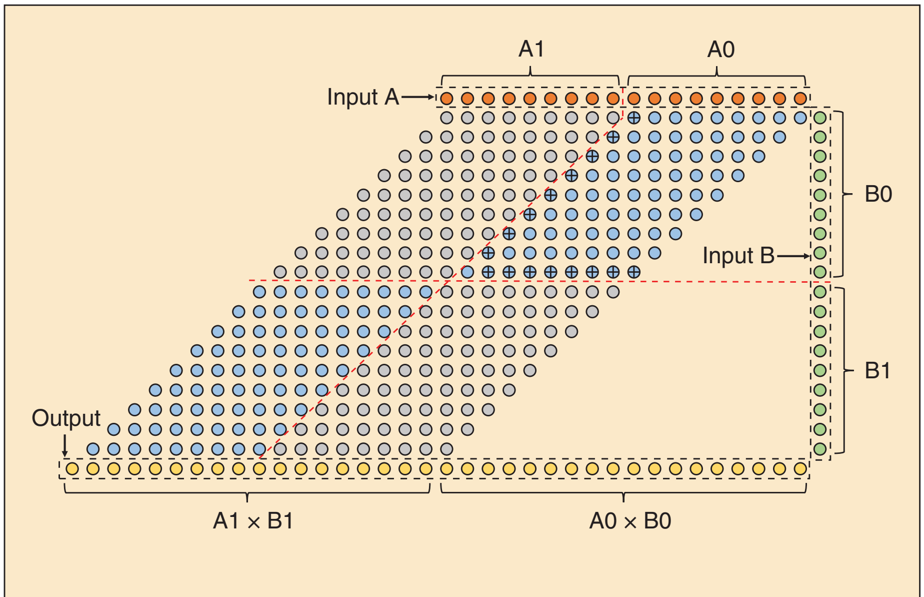


Figure 16. Fracturing an 18 × 18 multiplier array into two 9 × 9 arrays with the same number of input/output ports.

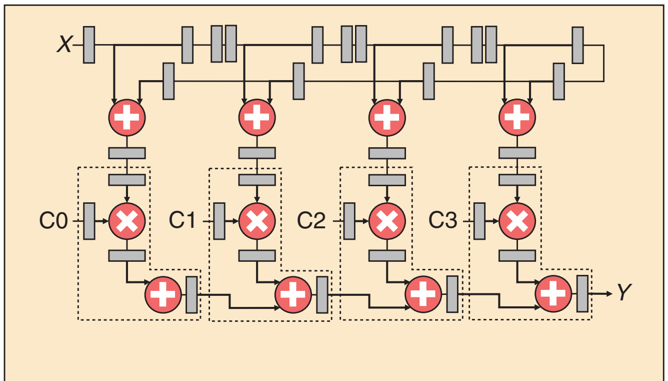


Figure 17. Systolic symmetric FIR filter circuit.

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. 17. 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 area efficiency gap increases to 3.9× and 8.5× for fixed and input coefficients, respectively. These gains are large but still less than the 35× gap between FPGAs and ASICs [11] 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 [11] are the interfaces to the programmable routing and the general inter-tile programmable routing wires and multiplexers 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 III.

The subsequent 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 key 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 (the key precision used in wireless base station signal processing) or one 27 × 27 multiplication (useful for single-precision floating-point mantissas). As a result, the simpler Stratix V DSP block spanned a single row, which is more friendly to Altera's row redundancy scheme. In addition, input pre-adders as well as embedded coefficient banks to store read-only filter weights were added [73], which allowed implementing the whole symmetric FIR filter structure shown in Fig. 17 inside the DSP blocks without the need for any soft logic resources. On the other hand, Xilinx switched from 18 × 18 multipliers to 25 × 18 multipliers in their Virtex-5 *DSP48E* tile [74], after which they incorporated input pre-adders and enhanced their adder/accumulator unit to also support bitwise logic operations in their Virtex-6 *DSP48E1* tile [75]. Then, they increased their multiplication width again to 27 × 18 bit and added a fourth input to their ALU in the Ultrascale family *DSP48E2* tile [76].

As illustrated in Fig. 15, 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 [77], [78]. More recently, FPGAs have been widely deployed in datacenters to accelerate various types of workloads such as search engines and network packet processing [9]. In addition, DL has emerged as a key component of many applications both in datacenter and edge workloads, with MAC being its core arithmetic operation. Driven by these new trends, 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 huge 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 [79]. 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% (i.e. 0.5% total die area

Table III.
Implementation results for a 51-tap symmetric 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 (input 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 (input coeff.)	12867	0	5.35 (8.5×)	217 (0.4×)

increase). Floating-point capabilities will also be supported in the *DSP58* tiles of the next-generation Xilinx Versal architecture [80].

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 [81]–[83]. 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, their energy efficiency compared to GPUs, and their lower development cost compared to custom ASICs. This has led both academic researchers and FPGA vendors to investigate adding native support for low-precision multiplication to DSP blocks. The authors of [84] enhance 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 [85] 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, Intel announced that the same `int9` mode of operation will be added to the next-generation Agilex DSP block along with half-precision floating-point (`fp16`) and brain float (`bfloat16`) precisions [86]. Also, the next-generation Xilinx Versal architecture will natively support `int8` multiplications in its *DSP58* tiles [80].

Throughout the years, the DSP block architecture has evolved to best suite 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 has recently announced an AI-optimized FPGA, the Stratix 10 NX, which replaces conventional DSP blocks with AI tensor blocks [87]. The new 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 [88]. 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 [89]. The next-generation Speedster7t FPGA from Achronix will also include a machine learning processing (MLP) block [90]. It supports a variety of precisions from `int16` down to `int3` in addition to `fp24`, `fp16` and `bfloat16` floating-point formats. The MLP block in Speedster7t will also feature a tightly coupled BRAM and circular register file that enable the reuse of both input values and output results. Each of these tightly integrated memory banks has a 72-bit external input but can be configured to have an up-to 144-bit output that feeds the MLP's multiplier arrays, reducing the number of required routing ports by 2×.

F. System-Level Interconnect: Network-on-Chip

FPGAs have continuously increased both in capacity and in the bandwidth of their external IO interfaces such as DDR, PCIe and Ethernet. Distributing the data traffic between these high-speed interfaces and the ever-larger soft fabric is a challenge. This system-level interconnect has traditionally been built by configuring parts of the FPGA logic and routing to implement *soft* buses that realize multiplexing, arbitration, pipelining and wiring between the relevant endpoints. These external interfaces operate at higher frequencies than the FPGA fabric can achieve, and therefore the only way to match their bandwidth is to use wider (soft) buses. 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 [91] 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 large 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 [92].

Abdelfattah and Betz [93]–[95] 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 [95]. 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 [95] 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. [96] 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 [9]. 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 [8]. Yazdanshenas et al. [97] showed that a hard NoC significantly improves resource utilization, operating frequency and routing congestion in such shell + role FPGA use cases. Other studies have proposed FPGA-specific optimizations to increase the area efficiency and performance of soft NoCs [98]–[100]. However, [101] shows that even optimized soft NoCs still trail hard NoCs in most respects (usable bandwidth, latency, area and routing congestion).

Recent Xilinx (Versal) and Achronix (Speedster7t) FPGAs integrate a hard NoC [102], [103] 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. 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. 18(a). The NoC has programmable routing tables that are configured at boot time and provides standard AXI interfaces [104] as its fabric ports. The Speedster7t NoC topology is optimized for external interface to

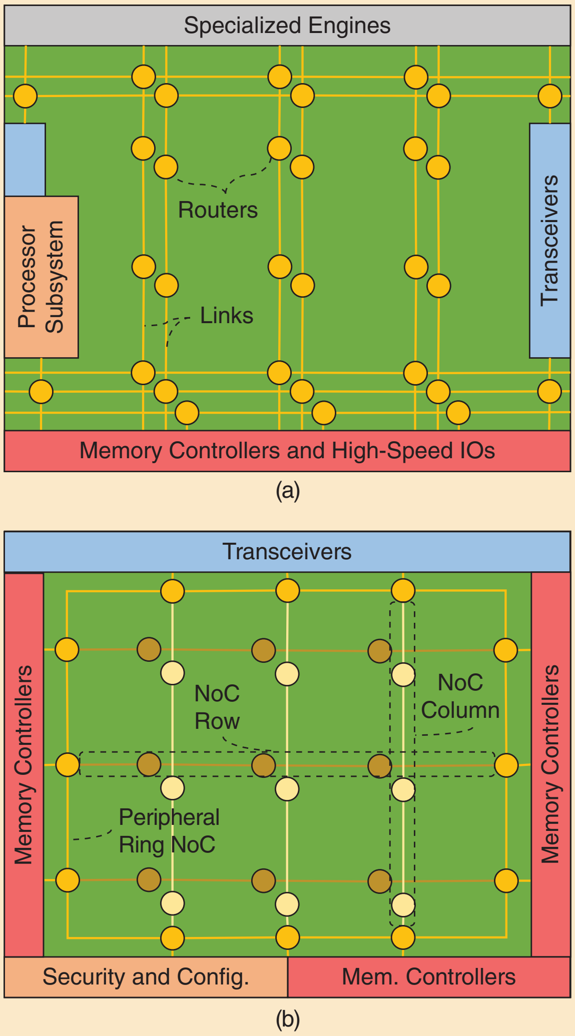


Figure 18. Network-on-Chip sytem-level interconnect in next-generation (a) Xilinx Versal and (b) Achronix Speedster7t architectures.

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. 18(b). 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.

G. Interposers

FPGAs have been early adopters of interposer technology that allows dense interconnection of multiple silicon dice. As shown in Fig. 19(a), a passive interposer is 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 2.5 D 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.

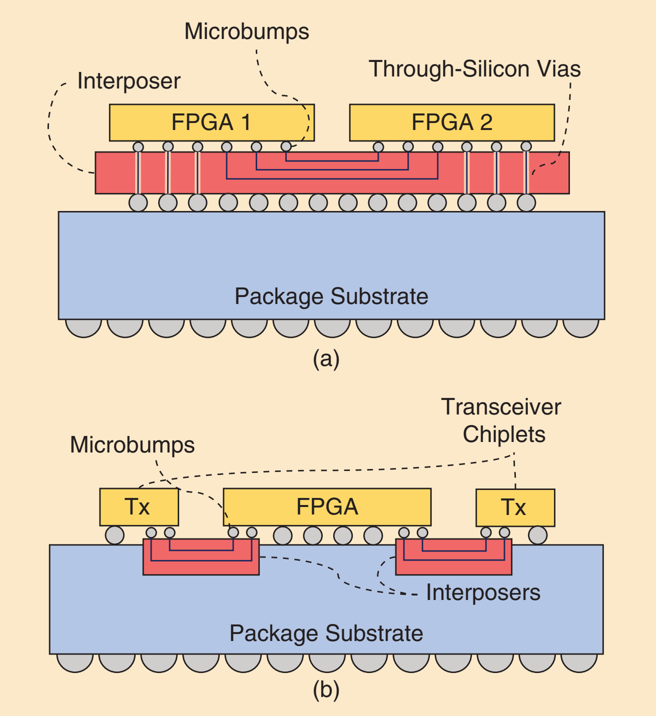


Figure 19. Different interposer technologies used for integrating multiple chips in one package in: (a) Xilinx multi-die interposer-based FPGAs and (b) Intel devices with EMIB-connected transceiver chiplets.

Xilinx's largest Virtex-7 (28 nm) and Virtex Ultrascale (20 nm) FPGAs use passive silicon interposers to integrate three to 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 sytem. For example, in Virtex-7 interposer-based FPGAs, only 23% of the vertical routing tracks cross between dice through the interposer [105], with an estimated additional delay of ~1 ns [106]. The study in [105] 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 next-generation Xilinx Versal architecture (discussed in Section III-F) 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.

Intel FPGAs instead use smaller interposers called embedded multi-die interconnect bridges (EMIB) carved into the package substrate as shown in Fig. 19(b). 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 [107]–[109] 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.

H. Other FPGA Components

Modern FPGA architectures contain other important components that we will not cover in detail.

One such component is the *configuration circuitry* that 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 or a hardened PCIe interface. 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 soft logic, allowing *partial reconfiguration* of one part of the device while another portion continues processing. 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, making a bitstream usable only by a single customer who purchases FPGAs with the proper key.

Since FPGA applications are often communicating with many different devices at different speeds, they commonly include dozens of clocks. 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 many high frequency clocks in different ways for different applications is challenging, and leads to special interconnect networks for clocks. These *clock networks* are similar in principle to the programmable interconnect of Section III-B but use routing wire and switch topologies that allow construction of low-skew networks like H-trees, and are implemented using wider metal and shielding conductors to reduce crosstalk and hence jitter.

IV. Conclusion and Future Directions

FPGAs have evolved from simple arrays of programmable logic blocks and IOs interconnected via programmable routing into more 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!

Acknowledgments

The authors would like to thank Fynn Schwiegelshohn for valuable feedback, and the NSERC/Intel industrial research chair in programmable silicon and the Vector institute for funding support.

Andrew Boutros received his B.Sc. degree in electronics engineering from the German University in Cairo in 2016, and his M.A.Sc. degree in electrical and computer engineering from the University of Toronto in 2018. He was a research scientist at Intel's Accelerator Architecture Lab in Oregon before he returned to the University of Toronto where he is currently pursuing his Ph.D. degree. His research interests include FPGA architecture and CAD, deep learning acceleration, and domain-specific architectures. He is a post-graduate affiliate of the Vector Institute for Artificial Intelligence and the Center for Spatial Computational Learning. He received two best paper awards at Reconfig 2016 and FPL 2018.

Vaughn Betz received his B.Sc. degree in electrical engineering from the University of Manitoba in 1991, his M.S. degree in electrical and computer engineering from the University of Illinois at Urbana–Champaign in 1993, and his Ph.D. degree in electrical and computer engineering from the University of Toronto in 1998. He is the original developer of the widely used VPR FPGA placement, routing and architecture evaluation CAD flow, and a lead developer in the VTR project that has built upon VPR. He co-founded Right Track CAD to commercialize VPR, and joined Altera upon its acquisition of Right Track CAD. Dr. Betz spent 11 years at Altera, ultimately as Senior Director of software engineering, and is one of the architects of the Quartus CAD system and the first five generations of the Stratix and Cyclone FPGA families. He is currently a professor and the NSERC/Intel Industrial Research Chair in Programmable Silicon at the University of Toronto. He holds 101 US patents and has published over 100 technical articles in the FPGA area, thirteen of which have won best or most significant paper awards. Dr. Betz is a Fellow of the IEEE and the NAI, and Faculty Affiliate of the Vector Institute for Artificial Intelligence.

References

[1] M. Hall and V. Betz, "HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs," 2020, arXiv:2007.10451.

[2] P. Yiannacouras et al., "Data parallel FPGA workloads: Software versus hardware," in *Proc. IEEE Int. Conf. Field-Programmable Logic Appl. (FPL)*, 2009, pp. 51–58.

[3] M. Cummings and S. Haruyama, "FPGA in the software radio," *IEEE Commun. Mag.*, vol. 37, no. 2, pp. 108–112, 1999. doi: 10.1109/35.747258.

[4] J. Rettkowski et al., "HW/SW co-design of the HOG algorithm on a Xilinx Zynq SoC," *J. Parallel Distrib. Comput.*, vol. 109, pp. 50–62, 2017. doi: 10.1016/j.jpdc.2017.05.005.

[5] A. Bitar et al., "Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2015, pp. 24–31.

[6] H. Krupnova and G. Saucier, "FPGA-based Emulation: Industrial and Custom Prototyping Solutions," in *Proc. Int. Workshop on Field-Programmable Logic Appl. (FPL)*, Springer-Verlag, 2000, pp. 68–77.

[7] A. Boutros et al., "Build fast, trade fast: FPGA-based high-frequency trading using high-level synthesis," in *Proc. IEEE Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, 2017, pp. 1–6.

[8] A. Putnam et al., "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. ACM/IEEE Int. Symp. Comput. Architecture (ISCA)*, 2014, pp. 13–24.

[9] A. M. Caulfield et al., "A cloud-scale acceleration architecture," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016, pp. 1–13.

[10] J. Fowers et al., "A configurable cloud-scale DNN processor for real-time AI," in *Proc. ACM/IEEE Int. Symp. Comput. Architecture (ISCA)*, 2018, pp. 1–14.

[11] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 203–215, 2007. doi: 10.1109/TCAD.2006.884574.

[12] A. Boutros et al., "You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018. doi: 10.1145/3242898.

[13] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*. Microelectronics Center of North Carolina, 1991.

[14] K. Murray et al., "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 13, no. 2, pp. 1–55, June 2020. doi: 10.1145/3388617.

[15] K. Murray et al., "Titan: enabling large and complex benchmarks in academic CAD," in *Proc. IEEE Int. Conf. Field-Programmable Logic Appl. (FPL)*, 2013, pp. 1–8.

[16] H. Parandeh-Afshar et al., "Rethinking FPGAs: Elude the flexibility excess of LUTs with and-inverter cones," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2012, pp. 119–128.

[17] H. Parandeh-Afshar et al., "Shadow AICs: Reaping the benefits of and-inverter cones with minimal architectural impact," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2013, pp. 279–279.

[18] H. Parandeh-Afshar et al., "Shadow and-inverter cones," in *Proc. IEEE Int. Conf. Field-Programmable Logic Appl. (FPL)*, 2013, pp. 1–4.

[19] G. Zgheib et al., "Revisiting and-inverter cones," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2014, pp. 45–54. doi: 10.1145/2554688.2554791.

[20] V. Betz et al., *Architecture and CAD for Deep-Submicron FPGAs*. Springer Science & Business Media, 1999.

[21] V. Betz and J. Rose, "How much logic should go in an FPGA logic block?" *IEEE Design Test Comput.*, vol. 15, no. 1, pp. 10–15, 1998. doi: 10.1109/54.655177.

[22] G. Lemieux et al., "Generating highly-routable sparse crossbars for PLDs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2000, pp. 155–164. doi: 10.1145/329166.329199.

[23] C. Chiasson and V. Betz, "COFFE: Fully-automated transistor sizing for FPGAs," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2013, pp. 34–41.

[24] E. Ahmed and J. Rose, "The effect of LUT and cluster size on deep-submicron FPGA performance and density," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 3, pp. 288–298, 2004. doi: 10.1109/TVLSI.2004.824300.

[25] "Stratix II Device Handbook, Volume 1 (SII5V1-4.5)." Altera Corp., 2007.

[26] D. Lewis et al., "The Stratix II logic and routing architecture," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2005, pp. 14–20.

[27] T. Ahmed et al., "Packing techniques for Virtex-5 FPGAs," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 2, no. 3, pp. 1–24, 2009. doi: 10.1145/1575774.1575777.

[28] W. Feng et al., "Improving FPGA performance with a S44 LUT structure," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2018, pp. 61–66. doi: 10.1145/3174243.3174272.

[29] "Versal ACAP Configurable Logic Block Architecture Manual (AM005 v1.0)," Xilinx Inc, 2020.

[30] D. Lewis et al., "Architectural enhancements in Stratix V," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2013, pp. 147–156.

[31] I. Ganusov and B. Devlin, "Time-borrowing platform in the Xilinx Ultrascale+ family of FPGAs and MPSoCs," in *Proc. IEEE Int. Conf.Field Programmable Logic Appl. (FPL)*, 2016, pp. 1–9.

[32] K. Murray et al., "Optimizing FPGA logic block architectures for arithmetic," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 6, pp. 1378–1391, 2020. doi: 10.1109/TVLSI.2020.2965772.

[33] S. Yazdanshenas and V. Betz, "Automatic circuit design and modelling for heterogeneous FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Technol. (ICFPT)*, 2017, pp. 9–16.

[34] J. Chromczak et al., "Architectural enhancements in Intel Agilex FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 140–149.

[35] S. Rasoulinezhad et al., "LUXOR: An FPGA logic cell architecture for efficient compressor tree implementations," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2020, pp. 161–171.

[36] A. Boutros et al., "Math doesn't have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 94–103.

[37] M. Eldafrawy et al., "FPGA logic block architectures for efficient deep learning inference," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 13, no. 3, pp. 1–34, 2020. doi: 10.1145/3393668.

[38] C. Chiasson and V. Betz, "Should FPGAs abandon the pass gate?" in *Proc. Int. Conf. Field-Programmable Logic Appl.*, 2013, pp. 1–8.

[39] FlexLogix eFPGA. https://flex-logix.com/efpga/

[40] V. Betz and J. Rose, "FPGA routing architecture: Segmentation and buffering to optimize speed and density," in *Proc. ACM Int. Symp. FPGAs*, 1999, pp. 59–68.

[41] O. Petelin and V. Betz, "The speed of diversity: Exploring complex FPGA routing toplogies for the global metal layer," in *Proc. IEEE Int. Conf. Field-Programmable Logic Appl. (FPL)*, 2016, pp. 1–10.

[42] D. Lewis et al., "The Stratix routing and logic architecture," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2003, pp. 12–20.

[43] X. Tang et al., "A study on switch block patterns for tileable FPGA routing architectures," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2019, pp. 247–250.

[44] G. Lemieux et al., "Directional and single-driver wires in FPGA interconnect," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2004, pp. 41–48.

[45] D. Lewis et al., "The Stratix 10 highly pipelined FPGA architecture," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2016, pp. 159–168.

[46] B. Gaide et al., "Xilinx adaptive compute acceleration platform: Versal architecture," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 84–93. doi: 10.1145/3289602.3293906.

[47] J. Tyhach et al., "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 *Proc. IEEE Custom Integrated Circuits Conf.*, 2004, pp. 431–434.

[48] N. Zhang et al., "Low-voltage and high-speed FPGA I/O cell design in 90nm CMOS," in *Proc. IEEE Int. Conf. ASIC*, 2009, pp. 533–536.

[49] T. Qian et al., "A 1.25Gbps programmable FPGA I/O buffer with multi-standard support," in *Proc. IEEE Int. Conf. Integr. Circuits Microsyst.*, 2018, pp. 362–365.

[50] P. Upadhyaya et al., "A fully-adaptive wideband 0.5–32.75Gb/s FPGA transceiver in 16nm FinFET CMOS technology," in *Proc. IIEEE Symp. VLSI Circuits*, 2016, pp. 1–2.

[51] "Implementing RAM functions in FLEX 10K Devices (A-AN-052-01)," Altera Corp., 1995.

[52] K. Tatsumura et al., "High density, low energy, magnetic tunnel junction based block RAMs for memory-rich FPGAs," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2016, pp. 4–11.

[53] T. Ngai et al., "An SRAM-programmable field-configurable memory," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, 1995, pp. 499–502.

[54] S. Wilton et al., "Architecture of centralized field-configurable memory," in *Proc. ACM Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 1995, pp. 97–103.

[55] S. Yazdanshenas et al., "Don't forget the memory: Automatic block RAM modelling, optimization, and architecture exploration," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 115–124.

[56] T. R. Halfhill, "Tabula's time machine," *Microprocessor Rep.*, vol. 131, 2010.

[57] "Mercury programmable logic device family (DS-MERCURY-2.2)," Altera Corp., 2003.

[58] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2010, pp. 41–50. doi: 10.1145/1723112.1723122.

[59] C. E. LaForest et al., "Multi-ported memories for FPGAs via XOR," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2012, pp. 209–218. doi: 10.1145/2145694.2145730.

[60] D. Lewis et al., "Architectural enhancements in Stratix-III and Stratix-IV," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2009, pp. 33–42.

[61] R. Tessier, et al. "Power-efficient RAM mapping algorithms for FPGA embedded memory blocks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 2, pp. 278–290, 2007. doi: 10.1109/TCAD.2006.887924.

[62] B.-C. C. Lai and J.-L. Lin, "Efficient designs of multiported memory on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 1, pp. 139–150, 2016. doi: 10.1109/TVLSI.2016.2568579.

[63] H. Wong et al., "Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2011, pp. 5–14. doi: 10.1145/1950413.1950419.

[64] E. Kadric et al., "Impact of memory architecture on FPGA energy consumption," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 146–155. doi: 10.1145/2684746.2689062.

[65] L. Ju et al., "NVM-based FPGA block RAM with adaptive SLC-MLC conversion," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2661–2672, 2018. doi: 10.1109/TCAD.2018.2857261.

[66] P. Longa and A. Miri, "Area-efficient FIR filter design on FPGAs using distributed arithmetic," in *Proc. IEEE Int. Symp. Signal Process. Inform. Technol.*, 2006, pp. 248–252.

[67] P. K. Meher et al., "FPGA realization of FIR Filters by efficient and flexible systolization using distributed arithmetic," *IEEE Trans. Signal Process.*, vol. 56, no. 7, pp. 3009–3017, 2008. doi: 10.1109/TSP.2007.914926.

[68] "Virtex-II platform FPGAs: Complete data sheet (DS031 v4.0)," Xilinx Inc, 2014.

[69] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C-22, no. 12, pp. 1045–1047, 1973. doi: 10.1109/T-C.1973.223648.

[70] "Using the DSP Blocks in Stratix & Stratix GX Devices (AN-214-3.0)," Altera Corp., 2002.

[71] "XtremeDSP for Virtex-4 FPGAs (UG073 v2.7)," Xilinx Inc, 2008.

[72] "DSP Blocks in Stratix III Devices (SIII51005-1.7)," Altera Corp., 2010.

[73] "Stratix V Device Handbook Volume 1: Device Interfaces and Integration (SV-5V1)," Altera Corp., 2020.

[74] "Virtex-5 FPGA XtremeDSP Design Considerations (UG193 v3.6)," Xilinx Inc, 2017.

[75] "Virtex-6 FPGA DSP48E1 Slice (UG369 v1.3)," Xilinx Inc, 2011.

[76] "UltraScale Architecture DSP Slice (UG579 v1.9)," Xilinx Inc, 2019.

[77] H. Parandeh-Afshar and P. Ienne, "Highly versatile DSP blocks for improved FPGA arithmetic performance," in *Proc. IEEE Int. Symp. Field-Programmable Custom Computing Mach. (FCCM)*, 2010, pp. 229–236.

[78] A. Cevrero et al., "Field programmable compressor trees: Acceleration of multi-input addition on FPGAs," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 2, no. 2, pp. 1–36, 2009. doi: 10.1145/1534916.1534923.

[79] M. Langhammer and B. Pasca, "Floating-point DSP block architecture or FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 117–125. doi: 10.1145/2684746.2689071.

[80] S. Ahmad et al., "Xilinx First 7nm device: Versal AI core (VC1902)," in *Proc. Hot Chips Symp.*, 2019, pp. 1–28.

[81] P. Gysel et al., "Hardware-oriented approximation of convolutional neural networks," 2016, arXiv:1604.03168.

[82] N. Mellempudi et al., "Mixed low-precision deep learning inference using dynamic fixed point," 2017, arXiv:1701.08978.

[83] A. Mishra et al., "WRPN: Wide reduced-precision networks," 2017, arXiv:1709.01134.

[84] A. Boutros et al., "Embracing diversity: Enhanced DSP blocks for low-precision deep learning on FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl. (FPL)*, 2018, pp. 35–357.

[85] S. Rasoulinezhad et al., "PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks," in *Proc. IEEE Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2019, pp. 35–44.

[86] Intel Corp. "Intel Agilex variable precision DSP blocks user guide (UG-20213)," Intel Corp., 2020.

[87] "Intel Stratix 10 NX FPGA: AI-optimized FPGA for high-bandwidth, low-latency AI acceleration (SS-1121-1.0)," Intel Corp., 2020.

[88] L. Gwennap, "Stratix 10 NX adds AI blocks," The Linley Group Newsletters, 2020.

[89] A. Boutros et al., "Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2020.

[90] "Speedster7t machine learning processing user guide (UG088)," Achronix Corp., 2019.

[91] "High Bandwidth Memory (HBM2) Interface Intel FPGA IP User Guide (UG-20031)," Intel Corp., 2020.

[92] M. T. Bohr, "Interconnect scaling: The real limiter to high performance ULSI," in *Proc. Int. Electron Devices Meeting*, 1995, pp. 241–244.

[93] M. S. Abdelfattah and V. Betz, "Design tradeoffs for hard and soft FPGA-based networks-on-chip," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2012, pp. 95–103.

[94] M. S. Abdelfattah and V. Betz, "The power of communication: Energy-efficient NoCs for FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl. (FPL)*, 2013, pp. 1–8.

[95] S. Abdelfattah and V. Betz, "The case for embedded networks on chip on field-programmable gate arrays," *IEEE Micro*, vol. 34, no. 1, pp. 80–89, 2013.

[96] M. S. Abdelfattah et al., "Take the highway: Design for embedded NoCs on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 98–107. doi: 10.1145/2684746.2689074.

[97] S. Yazdanshenas and V. Betz, "Quantifying and mitigating the costs of FPGA virtualization," in *Proc. IEEE Int. Conf. Field-Programmable Logic Appl. (FPL)*, 2017, pp. 1–7.

[98] N. Kapre, and and J. Gray, "Hoplite: A deflection-routed directional torus NoC for FPGAs," *ACM Trans. Reconfigurable Technol. Syst. (TRETS)*, vol. 10, no. 2, pp. 1–24, 2017. doi: 10.1145/3027486.

[99] M. K. Papamichael and J. C. Hoe, "CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2012, pp. 37–46.

[100] Y. Huan and A. DeHon, "FPGA optimized packet-switched NoC using split and merge primitives," in *Proc. IEEE Int. Conf. Field-Programmable Technol. (FPT)*, 2012, pp. 47–52.

[101] S. Yazdanshenas and V. Betz, "Interconnect solutions for virtualized field-programmable gate arrays," *IEEE Access*, vol. 6, pp. 10,497–10,507, 2018. doi: 10.1109/ACCESS.2018.2806618.

[102] I. Swarbrick et al., "Network-on-chip programmable platform in versal ACAP architecture," in *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 212–221.

[103] "Speedster7t network on chip user guide (UG089)," Achronix Corp., 2019.

[104] "AMBA AXI and ACE protocol specification," ARM Holdings, Tech. Rep., 2013.

[105] E. Nasiri et al., "Multiple dice working as one: CAD flows and routing architectures for silicon interposer FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 5, pp. 1821–1834, 2015. doi: 10.1109/TVLSI.2015.2478280.

[106] R. Chaware et al., "Assembly and reliability challenges in 3D integration of 28nm FPGA die on a large high density 65nm passive interposer," in *Proc. IEEE Electronic Components Technol. Conf.*, 2012, pp. 279–283.

[107] E. Nurvitadhi et al., "In-package domain-specific ASICs for intel Stratix 10 FPGAs: A case study of accelerating deep learning using tensortile ASIC," in *Proc. IEEE Int. Conf. Field-Programmable Logic Appl. (FPL)*, 2018, pp. 106–1064.

[108] E. Nurvitadhi et al., "Evaluating and Enhancing Intel Stratix 10 FPGAs for persistent real-time AI," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 119–119.

[109] E. Nurvitadhi et al., "Why compete when you can work together: FPGA-ASIC integration for persistent RNNs," in *Proc. IEEE Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM)*, 2019, pp. 199–207.