Chapter 1
FPGA Versus Software Programming: Why, When, and How?

Dirk Koch, Daniel Ziener, and Frank Hannig

This chapter provides background information for readers who are interested in the philosophy and technology behind FPGAs. We present this from a software engineer's viewpoint without hiding the hardware specific characteristics of FPGAs. Like it is very often not necessary to understand the internals of a CPU when developing software or applications (as in some cases a developer does not even know on which kind of CPU the code will be executed), this chapter should not be seen as compulsory. However, for performance tuning (an obvious reason for using accelerator hardware), some deeper background behind the used target technology is typically needed.

This chapter compromises on the abstraction used for the presentation and it provides a broader picture on FPGAs, including their compute paradigm, their underlying technology, and how they can be used and programmed. At many places, we have added more detailed information on FPGA technology and references to further background information. The presentation will follow a top-down approach and for the beginning, we can see an FPGA as a programmable chip that, depending on the device capacity, provides thousands to a few million bit-level processing elements (also called *logic cells*) that can implement any elementary logic function (e.g., any logic gate with a hand full inputs). These tiny processing elements are laid out regularly on the chip and a large flexible interconnection network (also called *routing fabric* with in some cases millions of programmable switches and wires

D. Koch (✉)
The University of Manchester, Manchester, UK
e-mail: dirk.koch@manchester.ac.uk

D. Ziener • F. Hannig
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: daniel.ziener@fau.de; frank.hannig@fau.de

D. Koch et al. (eds.), *FPGAs for Software Programmers*,
DOI 10.1007/978-3-319-26408-0_1

between these processing elements) allows to build more complex digital circuits from those logic cells. This allows implementing *any* digital circuit, as long as it fits the FPGA capacity.

1.1 Software and Hardware Programmability

Programmability allows it to adapt a device to perform different tasks using the same physical real estate (regardless of whether it is a CPU or FPGA). This is useful for *sharing resources* among different tasks and therefore for *reusing resources* over time, for *using the same device for various applications* (and hence allowing for lower device cost due to higher volume mass production), and for allowing *adaptations and customizations* in the field (e.g., fixing bugs or installing new applications).

1.1.1 Compute Paradigms

We all have heard about CPUs, GPUs, FPGAs and probably a couple of further ways to perform computations. The following paragraphs will discuss the distinct models, also known as *compute paradigms*, for the most popular processing options in more detail.

Traditional von Neumann Model

The most popular compute paradigm is the *von Neumann model* where a processor fetches instructions (stated by a program counter), decodes them, fetches operands, processes them by an Arithmetic logic unit (ALU) and writes them to a register file or memory. This takes (1) considerable resources (i.e. real estate) on the chip for the instruction fetch and decode units, (2) power these units take, and (3) I/O throughput for getting the instruction stream into the processor. This is known as the *von Neuman bottleneck* that in particular arises if instructions and data transfers to memory share the same I/O bus. This might need throttling down the machine because instructions cannot be fetched fast enough. All modern processors follow a *Harvard architecture* with separate caches for instructions and data which, however, still has many of the shortcomings of the von Neumann model. An illustration of the von Neumann model is given in Fig. 1.1a.

Vector Processing and SIMD

One major approach applied to many popular processors for improving the work done per instruction was adding vector units. As shown in Fig. 1.1b, a vector unit

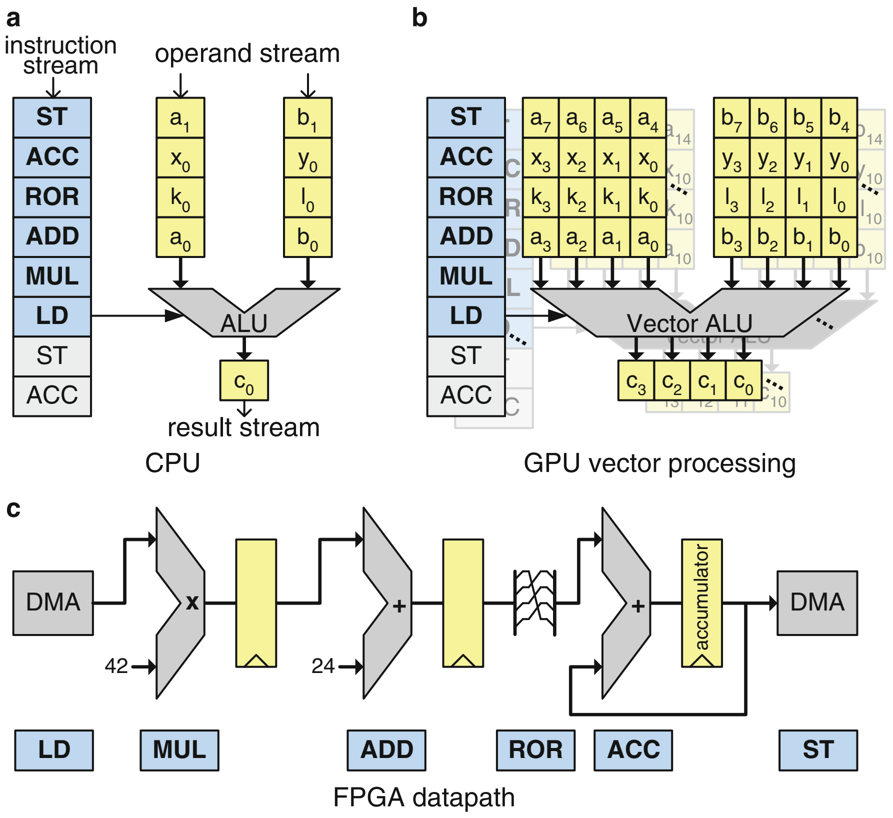


Fig. 1.1 Compute paradigms. (**a**) Traditional von Neumann model with a CPU decoding an instruction stream, fetching operands, processing them and writing results. (**b**) Vector processing achieves higher throughput by processing multiple values in parallel by sharing the same instruction stream (i.e. the same control flow). Multiple vector units can work in parallel on different data items, which is the model deployed in GPUs. (**c**) Reconfigurable datapath on an FPGA. Instead of performing different instructions on a single ALU, a chain of dedicated processing stages is used in a pipelined fashion. If resources and I/O throughput permit, the pipeline might also use vector processing or multiple pipelines may process in parallel

performs multiple identical operations on a set of input operands simultaneously, so the effort for instruction fetching and decoding is amortized on more actual processing work. Famous examples for vector extensions are the MMX technology introduced by Intel in their Pentium processors in 1997, followed by AMD with 3DNow! and ARM with the NEON vector extension in the ARM Cortex-A series. Vector units are also referred to as Single Instruction Multiple Data (SIMD) units. While vector processing is promising from an architectural point of view, it is challenging for compilers to harness this kind of computing and much of the code using vector units is written by hand using assembly language.

GPU Processing

For further improving processing performance (mostly driven by video processing for computer games), large amounts of lightweight vector processing elements have been integrated in Graphics Processor Units (GPUs). Here, lightweight means a simple instruction fetch and decode unit for controlling the vector units for improved performance-per-energy levels. However, communication between cores (or with the host machine) is done through global shared memory (i.e., each core can access any of the on-board GPU memory). As a consequence, with many hundreds of vector units on a single GPU, the required memory bandwidth of GPUs is enormous and despite a significant amount of on-chip caches. High-end GPUs factor this in by providing hundreds of gigabytes of memory throughput which takes a considerable amount of energy. For example, according to Vermij et al. [VFHB14], about half the energy is used by the shared memory, the L2 Cache, on-chip communication, and by the memory controllers over a series of benchmarks running on a GPU. As a reference, the corresponding power breakdown for a Xeon E5-2630 showed that "around 50% of the energy goes to the three levels of cache, and another 20% to 30% is spent on the memory controllers" [VFHB14].

GPU processing can work well on data parallel problems, like most graphics processing tasks that work on independent pixels or vertices, but problems that require computing a global result, synchronization will be needed as described in the following Sect. 1.1.2. For example in Fig. 1.1, the executed program is an unrolled loop (meaning we skipped any loop control instructions) that accumulates a result together. While we might be able to distribute the compute problem to parallel vector units, as illustrated in Fig. 1.1b (meaning that different processing elements compute a different part of the problem), we still have to accumulate a final result over all used vector units (see the `ACC` accumulate instruction in the examples that computes $R_O = R_O + R_I$). Depending on the problem, the synchronization effort can actually exceed the processing time and the GPU cores will spend more time on synchronization spinlocking than actually on providing useful work.

FPGA Datapath

As FPGAs can implement any digital circuit, they can implement architectures that follow the von Neuman, the vector, or the GPU model. This is in many cases a reasonable way to use FPGAs technology to implement another programmable architecture on top of the FPGA fabric. The FPGA will then mimic another programmable core (e.g., a well known CPU or a specialized programmable device) that is then programmed with rather conventional software developing techniques. This concept is known as *hardware overlays* and is covered in more detail in Chap. 15.6 for a systolic processor array. This is a computer architecture using small CPU-like processing elements that typically have little local memory and a next neighbor communication infrastructure. Note that the capacity of today's high density FPGAs allows easily to host several hundred 32-bit processors including a suitable infrastructure (with memories, on chip networks, and I/O peripherals) on a single device. A vector overlay is available with the MXP Matrix Processor from VectorBlox Computing Inc. That vector processor can deliver for some applications a 100× speedup over a baseline FPGA softcore processor and the design process is purely carried out using software programming.[1]

A way we can look at a processor is similar to having a universal worker who works off a job list (its program). However, if we take this analogy and compare it with how we perform industrial mass production, then we see that a different approach has been proven to perform much better: the *assembly belt*. The way an assembly belt production is orchestrated is to locate highly specialized and optimized workers (or machines) along a network of conveyor belts that move the product to produce (e.g., a car) and materials across the factory. This is exactly the idea behind the *FPGA datapath paradigm* where specialized processing elements form a processing pipeline and instead of material, data is streamed between and through these processing elements. As shown in Fig. 1.1c, there is no instruction stream and the original instruction sequence is decomposed in a chain of simple arithmetic operators. Consequently, there is no need for instruction fetch or decode, this is encoded directly into the structure of the datapath.

When arranging a factory for efficient production, this can include a very complex arrangement of workers, machines, and conveyor belts. Similarly, mapping certain computations onto an FPGA might need a complex arrangement of processing elements (or accelerator modules) with multiple processing pipelines that might split or join. Due to the enormous flexibility of the reconfigurable interconnection network provided by the FPGA, we can map virtually any communication dependency between processing elements (e.g., streaming, broadcast, multicast, bus-based communication, or even networks on chip). To summarize this: with FPGAs, we can tailor the machine to the problem, while with CPUs/GPUs we can only tailor the implementation to the machine.

While the clock frequency of such a processing pipeline on an FPGA is rather low with commonly only a few 100 MHz, we can start a new iteration in each operational cycle. A CPU, in contrast, has to rush through all instructions of a loop body before the next iteration can be started. So with longer pipelines, the speedup of a datapath implemented on an FPGA can be enormous. For example, the Maxeler OpenSPL compiler (which is introduced in Chap. 5) uses several of the High-Level Synthesis (HLS) techniques that are described in more detail in Chap. 2. Starting from a Java-like description, this compiler can automatically generate processing pipelines with eventually over a thousand pipeline stages on a single FPGA. And the Maxeler compiler is further able to exploit multiple chained FPGAs for implementing even larger pipelines.

As the clock frequency of processors is not likely to increase significantly and because the improvement in the micro architecture will probably not result in major performance increases, we cannot expect to see substantial single core

[1]http://vectorblox.com/about-technology/applications/.

(or thread) performance boosts in the near future. However, to fulfill demands for more performance, parallel execution has to be applied. In the last paragraph, it was described that parallel processing might often work well on GPUs for computing data parallel problems by distributing the problem to many GPU cores that then perform the same program but on different input data. This can be seen as *horizontal scaling* (or *horizontal parallelism*), because instead of running sequentially through a piece of code on one core, multiple cores are working side-by-side in this model. This can, of course, be applied to FPGAs. In addition, FPGAs allow for further *vertical scaling* (or *vertical parallelism*) where different processing elements perform different operations (typically in a pipelined fashion). If we assume, for example, a sequence of instructions in a loop body, it is not easy to parallelize the loop body by executing the first half of the instructions on one CPU and the other half of the instructions on another CPU. On an FPGA, however, this can be applied to any extend and all the way to individual instructions, as it is the case in an FPGA datapath. In other words, parallelizing an algorithm to FPGAs can in some cases be much easier performed than it would be possible for a multi core processors or GPUs. And with a rising demand for parallelizing algorithms it is more and more worth looking into FPGA programming.

1.1.2 Flexibility and Customization

Programming is used to solve specific tasks and the target platform (e.g., a processor or an FPGA) has to support running all tasks. We could say that virtually all available programmable platforms are Turing complete and therefore that we are (at least in theory) able to solve any task on virtually any programmable platform (assuming that we are not bound by memory). However, depending on the problem or algorithm that we want to solve, different programmable platforms are better suited than others. This is of course the reason for the large variety of programmable platforms. The following list states the benefits of some programmable architectures:

CPUs are specifically good when dealing with control-dominant problems. This is for example the case when some code contains many `if-then-else` or `case` statements. A lot of operating systems, GUIs, and all sorts of complex state machines fall into this category. Furthermore, CPUs are very adaptive and good at executing a large variety of different tasks reasonably well. Moreover, CPUs can change the executed program in just a couple of clock cycles (which is typically much less than a microsecond).

GPUs perform well on data-parallel problems that can be vectorized (for using the single instruction multiple data (SIMD) units available on the GPU thread processing elements) and for problems that need only little control flow and little synchronization with other threads or tasks. This holds in particular for floating-point computations, as heavily used in graphics processing. However for synchronization, typically spinlocking techniques are used that can take considerable performance and programming effort [RNPL15]

FPGAs achieve outstanding performance on stream processing problems and whenever pipelining can be applied on large data sets. FPGAs allow for a tight synchronized operation of data movement (including chip I/O) and processing of various different algorithms in a daisy-chained fashion. This often greatly reduces the need for data movement. For example, in a video processing system, we might decompress a video stream using only on-FPGA resources while streaming the result over to various further video acceleration modules without ever needing off-chip communication. This not only removes possible performance drops if we are I/O bound (e.g., if we have to occupy the memory subsystem for storing temporary data in a CPU-based system) but also helps in saving power. Please note that very often it takes much more power to move operands (or any kind of data) into a processing chip than performing computations on those operands.

Another important property of FPGAs is that they can implement any kind of register manipulation instruction directly. For example, most cryptographic algorithms require simple bit fiddling operations that can take a CPU tens to hundreds of cycles for performing all the needed bit-mask and shift operations (for, lets say, a 32-bit register value). On FPGAs however, each bit is directly accessible without any instruction and constant shift operations are implemented by the interconnection network and shift operations do not even take any FPGA logic resources (e.g., logic cells). In other words, FPGAs do not follow the restrictions of a pre-defined instruction set and computer organization. FPGAs allow us thinking in mathematical expressions and all sorts of operations rather than in a fixed register file and a collection of predefined instructions (as in all CPUs). For example, if we want to implement a modulo 10 Gray-code counter, we can build this directly in logic and use a small 4-bit register for storing the count state.

As can be seen, there is not a clear winning compute platform and depending on the problem and requirements, one or the other is better. In practice, there is also a human factor involved as it needs skilled people to program the different compute platforms. While there are many software developers, there are far less GPU or FPGA programmers, as can be observed on basically any job portal website. The intention of this book is to tackle exactly this issue and helping in making FPGAs accessible to software engineers.

A big difference in hardware and software programming platforms exists in the ability to control the structure and architecture of the programming platform. If we take a CPU or GPU, then a vendor (e.g., Intel, AMD, NVIDIA, or ARM) took decisions on the instruction set architectures, the organization and sizes of caches, and the capabilities of the I/O subsystem. In contrast when using FPGAs, there is much more freedom. If we need, for example, an operation for computing the correlation of a motion estimate vector, we can simply generate the corresponding logic and the correlation of one or many pixels can be computed in a single cycle, if needed. If an algorithm requires a certain amount of on-chip memory, we can allocate the exact corresponding number of memory blocks. For FPGAs, we are basically only limited by the available resources, but much less by architectural decisions of the FPGA vendor. And with devices providing millions of logic cells, thousands of arithmetic blocks, and tens of megabytes of on-chip memory, the limit is very low today.[2]

The need for flexibility can be discussed from a more system level perspective: with the exponential rise of capacity available on a single chip, we typically want to build more complex systems which implies the need for more heterogeneous compute platforms. This can be observed for the various compute resources including multicore CPUs, GPUs, and accelerators that are available in recent SoCs (System on Chips) targeting mobile phones or tablet PCs. However, by predefining an SoC architecture, we define its major characteristics and limit the level at which an equipment manufacturer can differentiate the final product. FPGAs, however, provide much more flexibility and they allow tailoring a large and complex SoC exactly to the needs of the application. This is relevant when considering very large chips in the future that should serve various different application domains. In this situation, FPGAs might utilize much better the available real estate on the chip due to their great customization abilities.

1.1.3 The Cost of Programmability

Unfortunately, programmability does not come for free. In a CPU, for example, we might have 100 possible instructions, but we will only start one machine instruction in each operational cycle, hence leaving probably most parts of the ALU underutilized.

Similarly on an FPGA, we typically cannot use all available functional blocks of the FPGA fabric in real-world applications and the routing between these blocks is carried out by a switchable interconnection network consisting of eventually many millions of programmable multiplexers that would not be needed when implementing the interconnection of a non-programmable digital circuit (i.e. an ASIC). In addition to real estate on the chip, all these switches will slow down the circuit and draw some extra power. All this would not be needed when connecting gates and functional blocks directly by wires, as it is the case for dedicated integrated circuits (i.e. ASICs).

This technology gap between FPGAs and ASICs was quantified by Kuon and Rose in [KR06]. Considering the same 90 nm fabrication process, an FPGA

[2]The limit is probably less what is available, but what is affordable. The high capacity flagship devices of the two major FPGA vendors Xilinx and Altera typically cost over 10 k US$. So the limit is often not a technological, but an economical one. However when Xilinx for example introduced its new Series-7 FPGAs, the smallest Kintex-7 device (XC7K70T) provided at a price tag below 100 US$ significantly more resources than its Virtex-2 flagship (XC2V8000) a decade ago. So in other words, what is far too expensive today might be well affordable very soon.

consisting of logic cells (based on lookup tables and flip-flops) would be on average 35× bigger (in terms of area) as the ASIC counterpart over a large set of examined benchmark circuits. However, by introducing additional dedicated functional blocks, such as multipliers and memory blocks, the gap in area can be reduced to ∼18×. For the performance it was found that FPGAs are 3–4× slower than their ASIC counterparts and they also draw about 10× more power.

However, due to high volume mass production, more advanced processes are commonly used that result in a lower monetary cost, better energy efficiency, and higher performance which all can offset much of the cost related to FPGA programmability. Furthermore, programmability removes underutilization if some tasks are used only very seldom. Due to re-programmability, the same physical resources can be shared and consequently resources can be better utilized as it would be possible with dedicated circuits for each task.[3] On a more macroscopic level, this shares some ideas behind cloud computing. The virtualization of servers in a cloud data center comes at a price that pays off because it commonly allows running a lower number of physical machines for a given load scenario.

The Impact of Technology Scaling

The probably most important driving factors in semiconductor technology today is *energy efficiency* and *power density*. We all know about the astonishing progress in silicon process technology which was firstly described by Moore's law [Moo65] that predicts an exponential growth of the transistor count on a chip (i.e. a doubling approximately every 18 months). Even given that chips for computing cannot be directly compared with memory chips, we can get a glimpse of what is possible today when calculating the number of transistors in a memory card that we plug into our phones, which can easily exceed a 100 billion transistors. However, while the density was (and still is) growing exponentially, power consumption per area stayed constant, which is known as Dennard scaling [DGnY$^+$74]. In other words, the energy efficiency per logic cell was also exponentially improving. Unfortunately,

[3]Most available FPGAs in these days can be partially reconfigured. This allows some parts of the FPGA to be operational (e.g., a CPU controlling and managing the system and a memory controller) while changing some accelerator modules. *Partial reconfiguration* can be used for a time-multiplexing of FPGA resources (e.g., if a problem does not fit the device). This helps for better utilizing an FPGA (and hence allowing for using a smaller and therefore cheaper and less power hungry device). This technique can also be used to speed-up computations. Then, instead of providing a chain of smaller modules concurrently on the FPGA, we might use reconfiguration to load only the module currently needed, such that we have more resources available for each module. This can then be used for exploiting a higher level of parallelization, which in turn can eventually speedup processing significantly. Finally, partial reconfiguration can be used for implementing dynamically instantiating hardware modules in a system (e.g., for creating a hardware thread at runtime). More comprehensive information on partial runtime reconfiguration is provided in [Koc13].

this has slowed down in recent years and the power density (i.e. power per surface) of many chips today is way beyond the power density of the hops that we use for cooking in our kitchens.

As the power density cannot exceed the thermal design of the chip and the surrounding system (the power used by a chip ends up in heat), the important question today is not only how many transistors can be squeezed on a chip, but how many of them are allowed to be powered and operational (or at what speed) without exceeding a given thermal budget. This phenomena is known as dark silicon [GHSV$^+$11]. However, if we are restricted by power rather than by transistor count on a chip, the cost of reconfigurability in terms of area is becoming a minor issue as all the unused resources make a programmable chip kind of dark anyway.

Economical Aspects

There are economical aspects that can pay for the extra cost of programmability. First of all, programmability allows faster time-to-market for products because being able to quickly develop, test, modify, and produce a product allows participating in higher profitable market windows. Programmability also allows for an after-sales business (e.g., by providing updates) and longer operation of equipment. The latter aspect was actually one of the major economical enablers for the success of FPGA technology. The internet hype and with it the fast development of technology and standards made it necessary to update network equipment in the field. However, for this application, CPUs had been far too slow and ASICs took too long to design and were not flexible enough. To overcome this, FPGAs had been (and are still) the ideal platform to power major parts of the internet infrastructure.

1.2 What Are FPGAs?

FPGAs (Field Programmable Gate Arrays) are digital chips that can be programmed for implementing arbitrary digital circuits. This means that FPGAs have first to be programmed with a so called *configuration* (often called a configuration bitstream) to set the desired behavior of the used functional elements of the FPGA. This can be compared to processors which are able to execute arbitrary programs after loading the corresponding machine code into some memory. Without loading any executable code, the processor is unable to perform any useful operation. Translated to FPGAs, an FPGA without a proper configuration does nothing. Only after the configuration step, the FPGA is able to operate with the desired behavior encoded into the configuration. FPGAs are an example for *PLDs* (Programmable Logic Devices). FPGAs have a significant market segment in the microelectronics and, particularly in the embedded system area. For example, FPGAs are commonly used in network equipment, avionics,[4] automotive, automation, various kinds of test equipment, medical devices, just to name some application domains. While currently only a tiny share of FPGAs are used for data processing in data centers, this is likely going to change in the near future as FPGAs not only provide very high performance, but they are also extremely energy efficient computing devices. This holds in particular when considering big data processing. For example, Microsoft recently demonstrated a doubling of the ranking throughput of their Bing search engine by equipping 1632 servers with FPGA accelerators which only added an extra 10 % in power consumption [PCC$^+$14]. In other words, Microsoft was able to improve the energy efficiency by 77 % while providing faster response times due to introducing FPGAs in their data centers.

Sometimes, FPGAs are referred to as *FPGA technology*. This expresses what target technology for implementing a digital circuit is used. In other words, it allows distinguishing if the target is an ASIC with logic gates as the building blocks or an FPGA that provides programmable logic cells. It is quite common to implement a digital circuit firstly on an FPGA for testing purposes before changing to ASIC technology when targeting high volume markets.

1.3 How FPGAs Work

So far we have talked mainly about programming an FPGA and what an FPGA is. This section will reveal its operation. Oversimplified, an FPGA consists of multiplexers and configuration memory cells that control those multiplexers and some wiring between them. This is illustrated in Fig. 1.2. It shows a multiplexer, the most important building block of an FPGA[5] and the operation of a multiplexer.

1.3.1 Lookup Table-Based Logic Cells

By connecting configuration memory cells to the select inputs of the multiplexer, a reconfigurable switch as part of a switch matrix is built, as shown in Fig. 1.2c. For implementing the actual reconfigurable logic cells, we connect configuration memory cells to the data inputs. For example, if we want to implement an AND gate, we will set the top configuration memory cell (connected to the input `11`) to `1`

[4]In an article from EETimes it is stated that "Microsemi already has over 1000 FPGAs in every Airbus A380" (http://www.electronics-eetimes.com/?cmp_id=7&news_id=222914228).

[5]Despite that FPGAs are basically made from multiplexers, it is a bit ironic that they are not very good at implementing multiplexers using their reconfigurable logic cells. In the case we want to implement multiplexers with many inputs, this would cost a considerable number of logic cells.

and the other three cells to `0`. Consequently, each logic cell provides a lookup table that stores the result for the desired Boolean function.

A lookup table (often abbreviated as *LUT*) can perform any Boolean function. The only limit is the size of the table, that is given by the number of inputs (with $k = 2$ in the figure). If larger functions have to be implemented than what fits in a single lookup table, the functions are implemented by cascading multiple logic cells (and therefore multiple lookup tables). For example, if we want to implement an AND gate with three instead of two inputs, then we implement the AND gate for two inputs, as described in the previous paragraph. The output is then connected to one input of another logic cell that again implements an AND gate. The other input of the second AND gate is then the third input of our 3-input AND gate.

The size of lookup tables found in commercial FPGAs varies from 3 to 6 inputs depending on the vendor and device family. Figure 1.3 gives more examples for truth tables of Boolean functions. For a table with k inputs, it takes 2^k configuration memory cells to build the logic cell. The examples in Fig. 1.3 show that the same logic function can be implemented by permuting inputs. This is used for simplifying the routing step (see Sect. 1.4.2). Instead of routing to a specific LUT input, it is sufficient to route to any input while eventually adjusting the table entries.

Usually, the lookup tables in the logic cells are combined with state flip-flops. These flip-flops are used storing states for FPGA-based circuits (e.g., the state of an n-bit counter can be stored in the corresponding flip-flops of n logic cells).

1.3.2 Configuration Memory Cells

Depending on the FPGA vendor, the configuration memory cells can be provided as SRAM cells, which allows fast and unlimited configuration data write processes. This is the technology used by the FPGA vendors Altera, Lattice, and Xilinx. SRAM-based FPGAs provide the highest logic densities and best performance. As SRAM cells are a volatile memory technology, it requires always a device configuration when powering the device up (which can be seen as a boot phase). The size of the configuration binary (i.e. the bitstream) can range from a few tens of kilobytes to a few tens of megabytes. As a consequence, the configuration of a large FPGA device can take over a second in some systems.

The FPGA vendor Microsemi provides FPGAs that are based on non-volatile configuration memory cells (Flash or antifuse). These devices are suited for extreme environments including space or avionics where electronic components are exposed to higher levels of ionizing radiation (which might impact the state of configuration memory cells of SRAM-based devices). In addition, these FPGAs are very power efficient, start immediately, and have better options for implementing cryptographic systems, because secrets can be stored persistent and securely on the FPGA device itself. However, the capacity and performance is lower than that of their SRAM counterparts.

1.3.3 Interconnection Network (Routing Fabric)

If we look at the technical parameters of an FPGA, we will find much information about the logic cells, including capacity, features, and performance. However, with respect to the area occupied on the FPGA die, the interconnection fabric with all its multiplexers is actually taking more resources than the logic cells themselves and this information is mostly hidden by the FPGA vendors. Also the performance of a circuit mapped to an FPGA is much related to the interconnection network. As a rule of thumb for present FPGAs, we can say that about 2/3 of the latency of a critical path is spent on reconfigurable interconnections. Note that the allowed clock frequency is the reciprocal of the *critical path delay*. This delay is the time needed for a signal to propagate from a flip-flop output through eventually multiple levels of logic cells and routing to a flip-flop input (this process has to be completed before the next clock edge arrives at the flip-flops). The critical path delay is then the slowest of all paths in a circuit mapped to an FPGA.

In practice, the multiplexers used for carrying out the switching between the logic cells will be much bigger than the ones shown in Fig. 1.2, with multiplexers typically providing between 10 and 30 inputs. Furthermore, the interconnection network provides additional switches for intermediate routing (and not only for connecting just the logic cell inputs). The FPGA vendors typically provide enough routing resources to carry out the routing of virtually any design. However, in some cases, it might be needed to work with lower utilization levels such that the ratio of routing resources per logic cells is higher which then permits routing the design.

1.3.4 Further Blocks

Today, most FPGAs provide not only logic cells, but a variety of building blocks (also called *primitives*). Please note that modules that are implemented in the FPGA fabric are often called *soft-logic* or *soft-IPs* (i.e. intellectual property cores implemented in the FPGA user logic). Respectively, cores that are provided with the FPGA fabric are called *hardened-logic* or *hard-IPs*.

Commonly provided hard-IPs on FPGAs include I/O blocks (e.g., for connections to high-speed transceivers which in some cases operate beyond 10 Gbit), dedicated memory blocks, and primitives providing multipliers (or even small ALUs). Furthermore, there could be special hardened functional blocks, like for example a PCIe connection core, DDR memory controllers, or even complete CPUs. Using hardened CPUs on an FPGA is covered in Sect. 15.2.3 (on page 268). The idea behind providing hardened IP blocks is their better area, performance and power ratio, when these blocks are used (because they do not need costly interconnection network). However, if a certain design leaves these blocks unused, they are basically wasted real estate (see also Sect. 1.1.3). FPGA vendors provide a selection of such blocks in order to meet the requirements of most customers.

1.3.5 How Much Logic is Needed to Implement Certain Logic Functions

When we develop an algorithm in software for a CPU, we are interested in its execution time and the memory requirements. In the case of FPGAs, we are not bound to a temporal domain only, but we have also a spatial domain, which means that we are also interested in how much real estate a certain logic function or algorithm will take. This is often not easy to answer and depends on many design factors. For example, a fully featured 32-bit processor requires about 2000 6-input lookup tables on a modern SRAM-based FPGA. However, there are area optimized 32 processors that use bit-serial processing and such a processor can be as small as 200 lookup tables [RVS$^+$10]. Such a processor would easily fit a few thousand times on a recent FPGA. In bit-serial operation, we are only computing a bit per clock cycle. Consequently, a 32-bit bit-serial processor is at least 32$\times$ slower than its parallel counterpart when operating at the same clock speed. This example shows that in hardware design it is often possible to trade processing speed for real estate. Some behavioral compilers incorporate this automatically for the developer. These tools allow specifying a certain compute throughput and the tool will minimize resource usage, or vice versa, it is possible to define the available resources and the tool maximizes performance.

For integer arithmetic, the cost for addition and subtraction scales linear with the size of the operands (in terms of bits) and is about one LUT per bit. Please note that it is common in hardware design to use arbitrarily sized bit vectors (e.g., there is no need to provide a 32 bit datapath when implementing only a modulo-10 counter). Comparators scale also linear with about 0.5 LUTs per bit to compare. Multipliers scale linear to quadratic, depending if we use sequential or full parallel operation. However, because multipliers can be expensive, there are multiplier blocks available on most FPGAs in these days. Division and square root is particularly expensive and should be avoided if possible. Floating point arithmetic is possible on FPGAs, but here addition and subtraction is more expensive due to the need for normalization (which is a comma shift operation that is needed if the exponents of two floating point numbers differ). However, many high-level synthesis tools support floating point data types and can exploit various optimizations for keeping implementation cost low. Many high level synthesis tools provide resource (and performance) estimators to give instantaneous feedback when programming for an FPGA.

This section is not meant to be complete, but is intended to give an introduction to how hardware can be made reprogrammable. Like with driving a car, where we do not have to understand thermodynamics, it is not necessary to understand these low-level aspects when programming an FPGA. The interested reader might find useful further information in the following books: [HD07, Bob07, Koc13].

1.4 FPGA Design Flow

The *FPGA design flow* comprises different *abstraction levels* which represent the description of an algorithm or an accelerator core over multiple transformation steps all the way to the final configuration bitstream to be sent to the FPGA. The design flow for FPGAs (and ASICs) can be divided into (1) a *fronted phase* and (2) a *backend phase*. The latter phase is typically carried out automatically by CAD tools (similar to the compilation process for a CPU). A short description about the backend flow is given in Sect. 2.5 on page 45 and more detailed further down in this section.

1.4.1 FPGA Front End Design Phase

Similar to the software world, there exists a wide variety of possibilities for generating a hardware description for FPGAs. All these approaches belong to the front end design phase and there exists a large ecosystem with languages, libraries, and tools targeting the front end design. One of the main aims of this book is to give a broad overview so that a software engineer can quickly narrow a search in this ecosystem.

Model-Driven and Domain-Specific Design

FPGAs can follow a model driven design approach which is commonly supported with comfortable and productive GUIs. For example, MATLAB Simulink from The MathWorks, Inc. and LabVIEW from National Instruments (see also Chap. 4) allow the generation of FPGA designs and running those designs pretty much entirely using a computer mouse only. These are out-of-the-box solutions targeting measurement instruments and control systems of virtually any complexity.

FPGA vendors and third-party suppliers provide large IP core libraries for various application domains. This will be covered in Chap. 15 for rapid SoC design using comfortable wizards.

Traditional Programming Languages

FPGAs can be programmed in traditional programming languages or in dialects of those. An overview of this is given in Chap. 3 along with more detailed examples in consecutive chapters. An example of a full environment with hardware platforms (from desktop systems to large data centers) and compilers for Java-like FPGA programming is provided in Chap. 5. The vast majority of high language design tools are for C/C++ or its dialects, which is covered in several chapters. This includes compilers from FPGA vendors in Chap. 6 (OpenCL from Altera Inc.) and Chap. 7 (Vivado HLS from Xilinx Inc.), as well as chapters on academic open source tools. The LegUp tool (developed at the University of Toronto) is presented in Chap. 10 and the ROCCC toolset (developed at The University of California, Riverside) is covered in Chap. 11, respectively.

Source-to-Source Compilation

For improving design productivity in specific domains (e.g., linear algebra), there exist source-to-source compilers that generate from an even higher abstraction level (commonly) C/C++ code that will then be further compiled by tools as described in the previous paragraph. Examples of this kind of solution are provided in Chap. 8 with the Merlin Compiler from Falcon Computing Solutions and in Chap. 12 with the HIPA^{cc} tool (developed at the Friedrich-Alexander University Erlangen-Nürnberg).

One big advantage of approaches operating at higher abstraction levels is not only that specifying a system is easier and can be done faster; there is also much less test effort needed. This is because many tools provide correct-by-construction transformations for specifications provided by the designer.

Low-Level Design

At the other end of the spectrum, there are the lower level Hardware Description Languages (HDLs) with Verilog and VHDL being the most popular ones. These languages give full control over the generated hardware. However, these languages are typically used by trained hardware design engineers and it needs a strong hardware background to harness the full potential of this approach. A solution that provides a higher level of abstraction (while still maintaining strong control over the generated hardware in an easier way than traditional HDLs) comes with the Bluespec SystemVerilog language and compiler, as revealed in Chap. 9.

The result of the front end design phase is a Register-transfer level (RTL) description that is passed to the backend flow. All this is described in more detail in the next section.

1.4.2 FPGA Backend Design Flow

The backend design flow for FPGAs and ASICs looks similar from a distance, as illustrated in Fig. 1.4. The figure shows a general design flow for electronic designs with FPGA and ASIC target technologies. The different abstraction levels were introduced by Gaijski et al. in [GDWL92] and include:

RTL level *The Register-Transfer Level* is basically an abstraction where all state information is stored in registers (or whatever kind of memory) and where logic (which includes arithmetic) between the registers is used to generate or compute new states. Consequently, the RTL level describes all memory elements (e.g., flip-flops, registers, or memories) and the used logic as well as the connections between the different memory and logic elements and therefore the flow of data through a circuit.

RTL specifications are commonly done using hardware description languages (HDLs). Those languages provide data types, logical and arithmetic operations, hierarchies, and many programming constructs (e.g., loops) that are known from software programming languages such as C. However, hardware is inherent parallel and there is not a single instruction executed at a point in time, but many modules that will work concurrently together in a system, which is well supported by HDLs.

We could see the RTL level in analogy to a CPU, where all state information is stored in registers (or some memory hierarchy) and where the ALU is in charge of changing the value of registers (or memories). The abstraction level of RTL is rather low and in an analogy to software, we could say it is comparable to an abstraction in the range somewhere from assembly to C. However, RTL code is still well readable and maintainable. As with assembly code, RTL code might be generated by other compilation processes as an intermediate representation and might not even be seen by an FPGA developer.

Logic level A synthesis tool will take the RTL description and translate it into a netlist which is the *logic abstraction level*. Here, any behavioral description from the RTL level is translated into registers and logic gates for implementing elementary Boolean functions. The netlist is a graph where the nodes denote registers and gates and the edges denote the connecting signals. For example, a compare between two registers A and B (let's say from an expression `if` A = B `then ...`) will result in an XNOR gate for each bit of the two values A and B (an XNOR is exactly `1` if both inputs are identical) and an AND gate (or a tree of AND gates) to check if all the XNOR gates deliver `1` at the output. This process is entirely carried out by logic compilers but can eventually be guided (hardware engineers prefer the term "to constrain a design or implementation") by the user (e.g., for either generating a small cost-efficient implementation or a high-performance implementation).

In the case of FPGAs as the synthesis target, there are specific synthesis options to control which kind of primitive on the FPGA fabric will later be used to implement a specific memory or piece of logic. For example, a shift register (which is basically a chain of flip-flops) can be implemented with the flip-flops after the lookup table function generators, with lookup table primitives that provide memory functionality, or with a dual-ported memory surrounded with some address counter logic. Similarly, multiplications can be done by dedicated multiplier blocks or by generating an array of bit-level multiplier cells (for example if we run short on multiplier blocks). In most cases, the default settings for the synthesis tool will work fine and many higher level tools are able to adjust the logic synthesis process automatically without bothering the designer.

Device Level The netlist from the previous level undergoes several optimization processes and is now to be implemented by the primitives (lookup tables, memory blocks, I/O cells, etc.) and the interconnection network of the FPGA.

In a first step, called *technology mapping*, the logic gates are fitted into the lookup tables. For this process, there exist algorithms for minimizing the number of lookup tables needed (e.g., Chortle [FRC90]). Alternatively there are algorithms for minimizing the depth of the tree when larger Boolean circuits have to be fitted into the given LUTs (e.g., FlowMap [CD94]). The last option results in fewer lookup tables to pass and consequently in lower propagation delay and therefore in a faster possible clock frequency.

After technology mapping, the primitives will get placed on the FPGA fabric. This is done with clustering algorithms and simulated annealing in order to minimize congestion and the distance between connected primitives.

The placed primitives get now routed and the difficulty here is that a physical wire of the FPGA can only be used exclusively for carrying out the linking of one signal path. A well-known algorithm for this is called Pathfinder [ME95]. In a nutshell, this algorithm routes each path individually and increases the cost for each wire segment according to how often it was used. Then by iteratively restarting the routing process, popular wires get more expensive which eventually resolves the conflict that multiple paths want to use the same wire.

The size of the place and route problem is quite large with more than one million primitives (i.e., logic cells, multipliers, memory blocks, etc.) and much more than ten million wires, each with about 10–30 possible programmable connections on recent devices. The tool time for computing all these steps can in some cases exceed a full day for such large devices and requires a few tens of gigabytes memory. Luckily, the process runs fully automated and there are techniques like incremental compilation and design preservation that allow keeping parts of the physical implementation untouched if only changes were done in a design. This will then significantly speed up design respins. In addition, there are tools that support a component-based design flow for plugging fully pre-implemented modules together to a working system. One approach for this method is known as hardware linking that works (in analogy to software linking) by stitching together configuration bitstreams of individual modules [KBT08]. This allows rapidly building systems without any time consuming logic synthesis, placement, and routing steps, but requires building a corresponding module library (e.g., [YKL15]).

The final mapped, placed, and routed netlist is then translated into a configuration bitstream that contains the settings for each primitive and switch matrix multiplexer (i.e. the values of the configuration memory cells shown in Fig. 1.2). More details on the low-level implementation can be found in [BRM99, Bob07, HD07, Koc13].

1.5 Overview

Most chapters in this book have already been introduced throughout this introduction. This introduction provides a discussion about compute paradigms and a brief introduction to FPGA technology. This is followed in Chap. 2 by a presentation of the theoretical background behind high-level synthesis which allows the generation of digital circuits directly from languages such as Java or C/C++. Chapter 3 gives a classification of various HLS approaches and Chaps. 4–12 are devoted to specific languages and tools including several case studies and small code examples.

Using FPGAs in a programming environment requires operating system services, some kind of infrastructure, and methods to couple this with a software subsystem. ReconOS (developed at the University of Paderborn) provides this functionality well for embedded systems, as presented in Chap. 13, while the LEAP FPGA operating system is targeting compute accelerators for x86 machines (Chap. 14).

The last two chapters are for software engineers who want to use FPGAs without designing any hardware by themselves. Chapter 15 provides examples on how complex systems can be built and programmed on FPGAs following a library approach. After this, Chap. 16 shows how FPGA technology can be used to host another programmable architecture that is directly software programmable. The overall goal of this book is to give software engineers a better understanding on accelerator technologies in general and FPGAs in particular. We also want to postulate the message that FPGAs are not only a vehicle for highly skilled hardware design engineers, but that there are alternative paths for software engineers to harness the performance and energy efficiency of FPGAs.

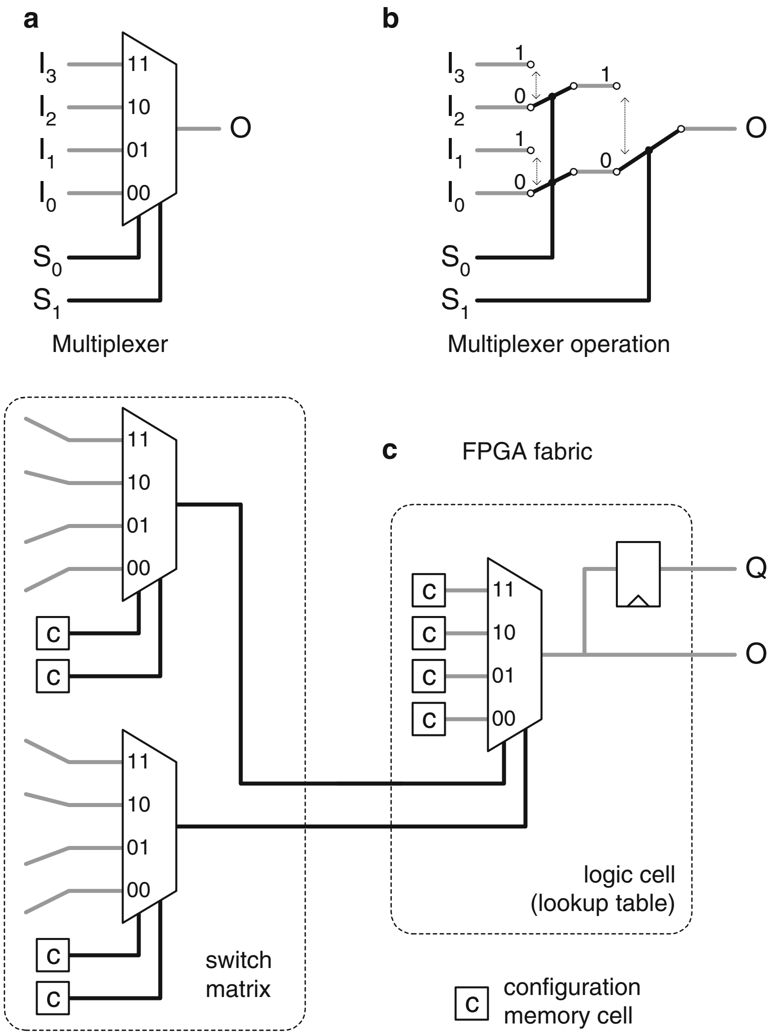


Fig. 1.2 FPGA fabric illustration. (**a**) The basic building block of an FPGA is a multiplexer. (**b**) Its internal operation can be compared with switches of a rail network and exactly one of the inputs (`I`) is connected with the output (`O`) depending on the select input (`S`). (**c**) (*left*) By connecting the select inputs to configuration memory cells, a switchable routing network is implemented for the interconnection between the logic cells. (**c**) (*right*) By connecting configuration memory cells to the multiplexer inputs, a lookup table is built that can implement any Boolean logic function for the lookup table inputs that are connected to the select inputs. For implementing larger combinatorial circuits, the `O` output can be connected to further logic cells and for storing states, a register is provided on the output `Q`. The values of the configuration memory cells are written during the configuration of the FPGA. By fitting large quantities of these basic blocks consisting of switch matrices and logic cells on a chip and connecting them together, we can build a reconfigurable FPGA fabric that can host any digital circuit

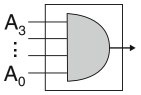
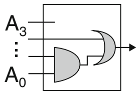
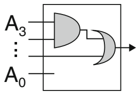
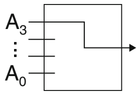
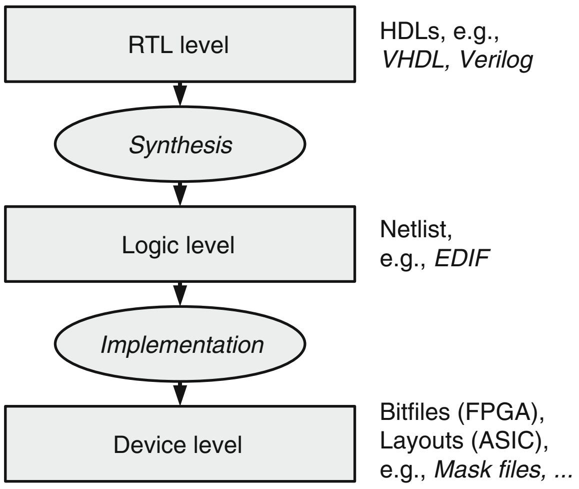
function	$A_0 \cdot A_1 \cdot A_2 \cdot A_3$	$(A_0 \cdot A_1) + A_2$	$(A_2 \cdot A_3) + A_1$	A_3
A_3 A_2 A_1 A_0				
0: 0 0 0 0	0	0	0	0
1: 0 0 0 1	0	0	0	0
2: 0 0 1 0	0	0	1	0
3: 0 0 1 1	0	1	1	0
4: 0 1 0 0	0	1	0	0
5: 0 1 0 1	0	1	0	0
6: 0 1 1 0	0	1	1	0
7: 0 1 1 1	0	1	1	0
8: 1 0 0 0	0	0	0	1
9: 1 0 0 1	0	0	0	1
A: 1 0 1 0	0	0	1	1
B: 1 0 1 1	0	1	1	1
C: 1 1 0 0	0	1	1	1
D: 1 1 0 1	0	1	1	1
E: 1 1 1 0	0	1	1	1
F: 1 1 1 1	1	1	1	1

Fig. 1.3 Lookup table configuration examples (partly taken from [Koc13])

Fig. 1.4 A general design flow for FPGA and ASIC designs with the synthesis and implementation steps and the different abstraction levels


RTL level
HDLs, e.g., *VHDL, Verilog*
Synthesis
Logic level
Netlist, e.g., *EDIF*
Implementation
Device level
Bitfiles (FPGA), Layouts (ASIC), e.g., *Mask files, ...*