3

Embedded Processors in FPGA Architectures

3.1 Introduction

Only 10 years ago we would have thought about the idea of a smart watch enabling us to communicate with a mobile phone, check our physical activity or heart rate, get weather forecast information, access a calendar, receive notifications, or give orders by voice as the subject of a futuristic movie. But, as we know now, smart watches are only one of the many affordable gadgets readily available in today’s market.

The mass production of such consumer electronics devices providing many complex functionalities comes from the continuous evolution of electronic fabrication technologies, which allows SoCs to integrate more and more powerful processing and communication architectures in a single device, as shown by the example in Figure 3.1.

FPGAs have obviously also taken advantage of this technological evolution. Actually, the development of FPSoC solutions is one of the areas (if not THE area) FPGA vendors have concentrated most of their efforts on over recent years, rapidly moving from devices including one general-purpose microcontroller to the most recent ones, which integrate up to 10 complex processor cores operating concurrently. That is, there has been an evolution from FPGAs with single-core processors to homogeneous or heterogeneous multicore architectures (Kurisu 2015), with symmetric multiprocessing (SMP) or asymmetric multiprocessing (AMP) (Moyer 2013).

This chapter introduces the possibilities FPGAs currently offer in terms of FPSoC design, with different hardware/software alternatives. But, first of all, we will discuss the broader concept of SoC and introduce the related terminology, which is closely linked to processor architectures.

From Chapter 1, generically speaking, a SoC can be considered to consist of one or more programmable elements (general-purpose processors, microcontrollers, DSPs, FPGAs, or application-specific processors) connected to and interacting with a set of specialized peripherals to perform a set of tasks. From this concept, a single-core, single-thread processor (general-purpose,
microcontroller, or DSP) connected to memory resources and specialized peripherals would usually be the best choice for embedded systems aimed at providing specific, non-time-critical functionalities. In these architectures, the processor acts as system master controlling data flows, although, in some cases, peripherals with memory access capabilities may take over data transfers with memory during some time intervals. Using FPGAs in this context provides higher flexibility than nonconfigurable solutions, because whenever a given software-implemented functionality does not provide good-enough timing performance, it can be migrated to hardware. In this solution, all hardware blocks are seen by the processor as peripherals connected to the same communication bus.

In order for single-core architectures to cope with continuous market demands for faster, more computationally powerful, and more energy-efficient solutions, the only option would be to increase operating frequency (taking advantage of nanometer-scale or 3D stacking technologies) and to reduce power consumption (by reducing power supply voltage). However, from the discussion in Chapter 1, it is clear that for the most demanding

FIGURE 3.1
Processing and communication features in a smart watch SoC.
current applications, this is not a viable solution, and the only ones that may work are those based on the use of parallelism, that is, the ability of a system to execute several tasks concurrently.

The straightforward approach to parallelism is the use of multiple single-core processors (with the corresponding multiple sources of power consumption) and the distribution of tasks among them so that they can operate concurrently. In these architectures, memory resources and peripherals are usually shared among the processors and all elements are connected through a common communication bus. Another possible solution is the use of multithreading processors, which take advantage of dead times during the sequential execution of programs (for instance, while waiting for the response from a peripheral or during memory accesses) to launch a new thread executing a new task. Although this gives the impression of parallel execution, it is just in fact multithreading. Of course, these two relatively simple (at least conceptually) options are valid for a certain range of applications, but they have limited applicability, for instance, because of interconnection delays between processors or saturation of the multithreading capabilities.

3.1.1 Multicore Processors

The limitations of the aforementioned approaches can be overcome by using multicore processors, which integrate several processor cores (either multithreading or not) on a single chip. Since in most processing systems the main factor limiting performance is memory access time, trying to achieve improved performance by increasing operating frequency (and, hence, power consumption) does not make sense above certain limits, defined by the characteristics of the memories. Multicore systems are a much more efficient solution than that because they allow tasks to be executed concurrently by cores operating at lower frequencies than those a single processor would require, while reducing communication delays among processors, all of them within the same chip. Therefore, these architectures provide a better performance–power consumption trade-off.

3.1.1.1 Main Hardware Issues

There are many concepts associated with multicore architectures, and the commercial solutions to tackle them are very diverse. This section concentrates just on the main ideas allowing to understand and assess the ability of FPGAs to support SoCs. Readers can easily find additional information in the specialized literature about computer architecture (Stallings 2016).

The first multicore processors date back some 15 years ago, when IBM introduced the POWER4 architecture (Tendler et al. 2002). The evolution since then resulted in very powerful processing architectures, capable of supporting different OSs on a single chip. One might think the ability to integrate multiple cores would have a serious limitation related to increased silicon area
and, in turn, cost. However, nanometer-scale and, more recently, 3D stacking technologies have enabled the fabrication of multicore chips at reasonably affordable prices. Today, one may easily find 16-core chips in the market.

As shown in Figure 3.2, multicore processors may be homogeneous (all of whose cores have the same architecture and instruction set) or heterogeneous (consisting of cores with different architectures and instruction sets). Most general-purpose multicore processors are homogeneous. In them, tasks (or threads) are interchangeable among processors (even at run time) with no effect on functionality, according to the availability of processing power in the different cores. Therefore, homogeneous solutions make an efficient use of parallelization capabilities and are easily scalable.

In spite of the good characteristics of homogeneous systems, there is a current trend toward heterogeneous solutions. This is mainly due to the very nature of the target applications, whose increasing complexity and growing need for the execution of highly specialized tasks require the use of platforms combining different architectures, as, for instance, microcontrollers, DSPs, and GPUs. Therefore, heterogeneous architectures are particularly suitable for applications where functionality can be clearly partitioned into specific tasks requiring specialized processors and not needing intensive communication among tasks.

Communications is actually a key aspect of any embedded system, but even more for multicore processors, which require low-latency, high-bandwidth communications not only between each processor and its memory/peripherals but also among the processors themselves. Shared buses may be used for this purpose, but the most current SoCs rely on crossbar interconnections (Vadja 2011). Given the importance of this topic, the on-chip buses most widely used in FPSoCs are analyzed in Section 3.5.

To reduce data traffic, multicore systems usually have one or two levels of local cache memory associated with each processor (so it can access the data it uses more often without affecting the other elements in the system),
plus one higher level of shared cache memory. A side benefit of using shared memory is that in case the decision is made to migrate some sequential programming to concurrent hardware or vice versa, the fact that all cores share a common space reduces the need for modifications in data or control structures. Examples of usual cache memory architectures are shown in Figure 3.3.

The fact that some data (shared variables) can be modified by different cores, together with the use of local cache memories, implicitly creates problems related to data coherence and consistency. In brief, coherence means all cores see any shared variable as if there were no cache memories in the system, whereas consistency means instructions to access shared variables are programmed in the sharing cores in the right order. Therefore, coherence is an architectural issue (discussed in the following) and consistency a programming one (beyond the scope of this book).

A multicore system using cache memories is coherent if it ensures all processors sharing a given memory space always “see” at any position within it the last written value. In other words, a given memory space is coherent if a core reading a position within it retrieves data according to the order the cores sharing that variable have written values for it in their local caches. Coherence is obviously a fundamental requirement to ensure all processors access correct data at any time. This is the reason why all multicore processors include a cache-coherent memory system.

Although there are many different approaches to ensure coherence, all of them are based on modification–invalidation–update mechanisms. In a simplistic way, this means that when a core modifies the value of a shared variable in its local cache, copies of this variable in all other caches are invalidated and must be updated before they can be used.
3.1.1.2 Main Software Issues

As in the case of hardware, there are many software concepts to be considered in embedded systems, and multicore ones in particular, at different levels (application, middleware, OS) including, but not limited to, the necessary mechanisms for multithreading control, partitioning, resource sharing, or communications.

Different scenarios are possible depending on the complexity of the software to be executed by the processor and that of the processor itself,* as shown in Figure 3.4. For simple programs to be executed in low-end processors, the usual approach is to use bare-metal solutions, which do not require any software control layer (kernel or OS). Two intermediate cases are the implementation of complex applications in low-end processors or simple applications in high-end processors. In both cases, it is usual (and advisable) to use at least a simple kernel. Although this may not be deemed necessary for the latter case, it is highly recommended for the resulting system to be easily scalable. Finally, in order to efficiently implement complex applications in high-end processors, a real-time or high-end OS is necessary (Walls 2014). Currently, this is the case for most embedded systems.

Other important issues to be considered are the organization of shared resources, task partitioning and sequencing, as well as communications between tasks and between processors. From the point of view of the software architecture, these can be addressed by using either AMP or SMP approaches, depicted in Figure 3.5.

SMP architectures apply to homogeneous systems with two or more cores sharing memory space. They are based on using only one OS (if required) for

![FIGURE 3.4](image)

Software scenarios.

---

* Just to have a straightforward idea about complexity, we label as low-end processors those whose data buses are up to 16-bit wide and as high-end processors those with 32-bit or wider data buses.
all cores. Since the OS has all the information about the whole system hardware at any point, it can efficiently perform a dynamic distribution of the workload among cores (which implies extract application parallelism, partition of tasks/threads, and dynamic assignment of tasks to cores), as well as the control of the ordering of task completion and of resource sharing among cores. Resource sharing control is one of the most important advantages of SMP architectures. Another significant one is easy interprocess communication, because there is no need for implementing any specific communication protocol, thus avoiding the overhead this would introduce. Finally, debugging tasks are simpler when working with just one OS.

SMP architectures are clearly oriented to get the most possible advantage of parallelism to maximize processing performance, but they have a main limiting factor, related to the dynamic distribution of workload. This factor affects the ability of the system to provide a predictable timing response, which is a fundamental feature in many embedded applications. Another past drawback, the need for an OS supporting multicore processing, is not a significant problem anymore given the wide range of options currently available (Linux, embedded Windows, and Android, to cite just some).

In contrast to SMP, AMP architectures can be implemented in either homogeneous or heterogeneous multicore processors. In this case, each core runs its own OS (either separate copies of the same or totally different ones; some cores may even implement a bare-metal system). Since none of the OSs is specifically in charge of controlling shared resources, such control must be very carefully performed at the application level. AMP solutions are oriented to applications with a high level of intrinsic parallelism, where critical tasks
are assigned to specific resources in order for a predictable behavior to be achieved. Usually, in AMP systems, processes are locked (assigned) to a given processor. This simplifies the individual control of each core by the designer. In addition, it eases migration from single-core solutions.

### 3.1.2 Many-Core Processors

Single- and multicore solutions are the ones most commonly found in SoCs, but there is a third option, many-core processors, which find their main niche in systems requiring a high scalability (mostly intensive computing applications), for instance, cloud computing datacenters. Many-core processors consist of a very large number of cores (up to 400 in some existing commercially available solutions [Nickolls and Dally 2010; NVIDIA 2010; Kalray 2014]), but are simpler and have less computing power than those used in multicore systems. These architectures aim at providing massive concurrency with a comparatively low energy consumption. Although many researchers and vendors (Shalf et al. 2009; Jeffers and Reinders 2015; Pavlo 2015) claim this will be the dominant processing architecture in the future, its analysis is out of the scope of this book, because until now, it has not been adopted in any FPGA.

### 3.1.3 FPSoCs

At this point two pertinent questions arise: What is the role of FPGAs in SoC design, and what can they offer in this context? Obviously, when speaking of parallelism or versatility, no hardware platform compares to FPGAs. Therefore, combining FPGAs with microcontrollers, DSPs, or GPUs clearly seems to be an advantageous design alternative for a countless number of applications demanded by the market. Some years ago, FPGA vendors realized the tremendous potential of SoCs and started developing chips that combined FPGA fabric with embedded microcontrollers, giving rise to FPSoCs.

The evolution of FPSoCs can be summarized as shown in Figure 3.6. Initially, FPSoCs were based on single-core soft processors, that is, configurable microcontrollers implemented using the logic resources of the FPGA fabric. The next step was the integration in the same chip as the FPGA fabric of single-core hard processors, such as PowerPC. In the last few years, several families of FPGA devices have been developed that integrate multicore processors (initially homogeneous architectures and, more recently, heterogeneous ones). As a result, the FPSoC market now offers a wide portfolio of low-cost, mid-range, and high-end devices for designers to choose from depending on the performance level demanded by the target application.

FPGAs are among the few types of devices that can take advantage of the latest nanometer-scale fabrication technologies. At the time of writing this book, according to FPGA vendors (Xilinx 2015; Kenny 2016), high-end FPGAs are fabricated in 14 nm process technologies, but new families have already
been announced based on 10 nm technologies, whereas the average for ASICs is 65 nm. The reason for this is just economic viability. When migrating a chip design to a more advanced node (let us say from 28 to 14 nm), the costs associated with hardware and software design and verification dramatically grow, to the extent that for the migration to be economically viable, the return on investment must be in the order of hundreds of millions of dollars. Only chips for high-volume applications or those that can be used in many different applications (such as FPGAs) can get to those figures.

The different FPSoC options currently available in the market are analyzed in the following sections.

### 3.2 Soft Processors

As stated in Section 3.1.3, soft processors are involved in the origin of FPSoC architectures. They are processor IP cores (usually general-purpose ones) implemented using the logic resources of the FPGA fabric (distributed logic, specialized hardware blocks, and interconnect resources), with the advantage of having a very flexible architecture.
As shown in Figure 3.7, a soft processor consists of a processor core, a set of on-chip peripherals, on-chip memory, and interfaces to off-chip memory. Like microcontroller families, each soft processor family uses a consistent instruction set and programming model.

Although some of the characteristics of a given soft processor are pre-defined and cannot be modified (e.g., the number of instruction and data bits, instruction set architecture [ISA], or some functional blocks), others can be defined by the designer (e.g., type and number of peripherals or memory map). In this way, the soft processor can, to a certain extent, be tailored to the target application. In addition, if a peripheral is required that is not available as part of the standard configuration possibilities of the soft processor, or a given available functionality needs to be optimized (for instance, because of the need to increase processing speed in performance-critical systems), it is always possible for the designer to implement a custom peripheral using available FPGA resources and connect it to the CPU in the same way as any “standard” peripheral.

The main alternative to soft processors are hard processors, which are fixed hardware blocks implementing specific processors, such as the ARM’s Cortex-A9 (ARM 2012) included by Altera and Xilinx in their latest families of devices. Although hard processors (analyzed in detail in Section 3.3) provide some advantages with regard to soft ones, their fixed architecture causes not all their resources to be necessary in many applications, whereas in other cases there may not be enough of them. Flexibility then becomes the main advantage of soft processors, enabling the development of custom solutions to meet performance, complexity, or cost requirements. Scalability and reduced risk of obsolescence are other significant advantages of soft processors. Scalability refers to both the ability of adding resources to support new features or update existing ones along the whole lifetime of the system and the
possibility of replicating a system, implementing more than one processor in the same FPGA chip. In terms of reduced risk of obsolescence, soft processors can usually be migrated to new families of devices. Limiting factors in this regard are that the soft processor may use logic resources specific to a given family of devices, which may not be available in others, or that the designer is not the actual owner of the HDL code describing the soft processor.

Soft processor cores can be divided into two groups:

1. Proprietary cores, associated with an FPGA vendor, that is, supported only by devices from that vendor.
2. Open-source cores, which are technology independent and can, therefore, be implemented in devices from different vendors.

These two types of soft processors are analyzed in Sections 3.2.1 and 3.2.2, respectively. Although there are many soft processors with many diverse features available in the market, without loss of generality, we will focus on the main features and the most widely used cores, which will give a fairly comprehensive view of the different options available for designers.

### 3.2.1 Proprietary Cores

Proprietary cores are optimized for a particular FPGA architecture, so they usually provide a more reliable performance, in the sense that the information about processing speed, resource utilization, and power consumption can be accurately determined, because it is possible to simulate their behavior from accurate hardware models. Their major drawback is that the portability of and the possibility of reusing the code are quite limited.

Open-source cores are portable and more affordable. They are relatively easy to adapt to different FPGA architectures and to modify. On the other hand, not being optimized for any particular architecture, usually, their performance is worse and less predictable, and their implementation requires more FPGA resources to be used.

Xilinx’s PicoBlaze (Xilinx 2011a) and MicroBlaze (Xilinx 2016a) and Altera’s Nios-II* (Altera 2015c), whose block diagrams are shown in Figure 3.8a through c, respectively, have consistently been the most popular proprietary processor cores over the years. More recently, Lattice Semiconductor released the LatticeMico32 (LM32) (Lattice 2012) and LatticeMico8 (LM8) (Lattice 2014) processors† whose block diagrams are shown in Figure 3.8d and e, respectively.

---

* Altera previously developed and commercialized the Nios soft processor, predecessor of Nios-II.
† Although LM8 and LM32 are actually open-source, free IP cores, since they are optimized for Lattice FPGAs, they are better analyzed together with proprietary cores.
PicoBlaze and LM8 are 8-bit RISC microcontroller cores optimized for Xilinx* and Lattice FPGAs, respectively. Both have a predictive behavior, particularly PicoBlaze, all of whose instructions are executed in two clock cycles. Both have also similar architectures, including:

* KCPSM3 is the PicoBlaze version for Spartan-3 FPGAs, and KCPSM6 for Spartan-6, Virtex-6, and Virtex-7 Series.
Embedded Processors in FPGA Architectures

- General-purpose registers (16 in PicoBlaze, 16 or 32 in LM8).
- Up to 4 K of 188-bit-wide instruction memory.
- Internal scratchpad RAM memory (64 bytes in PicoBlaze, up to 4 GB in 256-byte pages in LM8).
- Arithmetic Logic Unit (ALU).
- Interrupt management (one interrupt source in PicoBlaze, up to 8 in LM8).

The main difference between PicoBlaze and LM8 is the communication interface. None of it includes internal peripherals, so all required peripherals must be separately implemented in the FPGA fabric. PicoBlaze communicates with them through up to 256 input and up to 256 output ports, whereas LM8 uses a Wishbone interface from OpenCores, described in Section 3.5.4.

Similarly, although MicroBlaze, Nios-II, and LM32 are also associated with the FPGAs of their respective vendors, they have many common characteristics and features:

- 32-bit general-purpose RISC processors.
- 32-bit instruction set, data path, and address space.
- Harvard architecture.
- Thirty-two 32-bit general-purpose registers.
- Instruction and data cache memories.
- Memory management unit (MMU) to support OSs requiring virtual memory management (only in MicroBlaze and Nios-II).
- Possibility of variable pipeline, to optimize area or performance.
- Wide range of standard peripherals such as timers, serial communication interfaces, general-purpose I/O, SDRAM controllers, and other memory interfaces.
- Single-precision floating point computation capabilities (only in MicroBlaze and Nios-II).
- Interfaces to off-chip memories and peripherals.
- Multiple interrupt sources.
- Exception handling capabilities.
- Possibility for creating and adding custom peripherals.
- Hardware debug logic.
- Standard and real-time OS support: Linux, μCLinux, MicroC/OS-II, ThreadX, eCos, FreeRTOS, uC/OS-II, or embOS (only in MicroBlaze and Nios-II).
A soft processor is designed to support a certain ISA. This implies the need for a set of functional blocks, in addition to instruction and data memories, peripherals, and resources, to connect the core to external elements. The functional blocks supporting the ISA are usually implemented in hardware, but some of them can also be emulated in software to reduce FPGA resource usage. On the other hand, not all blocks building up the core are required for all applications. Some of them are optional, and it is up to the designer whether to include them or not, according to system requirements for functionality, performance, or complexity. In other words, a soft processor core does not have a fixed structure, but it can be adapted to some extent to the specific needs of the target application.

Most of the remainder of this section is focused on the architecture of the Nios-II soft processor core as an example, but a vast majority of the analyses are also applicable to any other similar soft processors. As shown in Figure 3.8c, the Nios-II architecture consists of the following functional blocks:

- **Register sets**: They are organized in thirty-two 32-bit general-purpose registers and up to thirty-two 32-bit control registers. Optionally, up to 63 shadow register sets may be defined to reduce context switch latency and, in turn, execution time.

- **ALU**: It operates with the contents of the general-purpose registers and supports arithmetic, logic, relational, and shift and rotate instructions. When configuring the core, designers may choose to have some instructions (e.g., division) implemented in hardware or emulated in software, to save FPGA resources for other purposes at the expense of performance.

- **Custom instruction logic (optional)**: Nios-II supports the addition of not only custom components but also of custom instructions, for example, to accelerate algorithm execution. The idea is for the designer to be able to substitute a sequence of native instructions by a single one executed in hardware. Each new custom instruction created generates a logic block that is integrated in the ALU, as shown in Figure 3.9. This is an interesting feature of the Nios-II architecture not provided by others.

Up to 256 custom instructions of five different types (combinational, multicycle, extended, internal register file, and external interface) can be supported. A combinational instruction is implemented through a logic block that performs its function within a single clock cycle, whereas multicycle (sequential) instructions require more than one clock cycle to be completed. Extended instructions allow several (up to 256) combinational or multicycle instructions to be implemented in a single logic block. Internal register file custom instructions are those that can operate with the internal registers of their logic block instead of with Nios-II general-purpose registers (the ones used by other custom instructions and by native instructions).
Finally, external interface custom instructions generate communication interfaces to access elements outside of the processor’s data path.

Whenever a new custom instruction is created, a macro is generated that can be directly instantiated in any C or C++ application code, eliminating the need for programmers to use assembly code (they may use it anyway if they wish) to take advantage of custom instructions.

In addition to user-defined instructions, Nios-II offers a set of predefined instructions built from custom instruction logic. These include single-precision floating-point instructions (according to IEEE Std. 754-2008 or IEEE Std. 754-1985 specifications) to support computation-intensive floating-point applications:

- **Exception controller**: It provides response to all possible exceptions, including internal hardware interrupts, through an exception handler that assesses the cause of the exception and calls the corresponding exception response routine.

- **Internal and external interrupt controller (EIC) (optional)**: Nios-II supports up to 32 internal hardware interrupt sources, whose priority is determined by software. Designers may also create an EIC and connect it to the core through an EIC interface. When using EIC, internal interrupt sources are also connected to it and the internal interrupt controller is not implemented.

- **Instruction and data buses**: Nios-II is based on a Harvard architecture. The separate instruction and data buses are both implemented using 32-bit Avalon-MM master ports, according to Altera’s proprietary Avalon interface specification. The Avalon bus is analyzed in Section 3.5.2.
The data bus allows memory-mapped read/write access to both data memory and peripherals, whereas the instruction bus just fetches (reads) the instructions to be executed by the processor. Nios-II architecture does not specify the number or type of memories and peripherals that can be used, nor the way to connect to them either. These features are configured when defining the FPSoC. However, most usually, a combination of (fast) on-chip embedded memory, slower off-chip memory, and on-chip peripherals (implemented in the FPGA fabric) is used.

- **Instruction and data cache memories** *(optional)*: Cache memories are supported in the instruction and data master ports. Both instruction and data caches are an intrinsic part of the core, but their use is optional. Software methods are available to bypass one of them or both. Cache management and coherence are managed in software.

- **Tightly coupled memories** *(TCM)* *(optional)*: The Nios-II architecture includes optional TCM ports aimed at ensuring low-latency memory access in time-critical applications. These ports connect both instruction and data TCMs, which are on chip but external to the core. Several TCMs may be used, each one associated with a TCM port.

- **MMU** *(optional)*: This block handles virtual memory, and, therefore, its use makes only sense in conjunction with an OS requiring virtual memory. Its main tasks are memory allocation to processes, translation of virtual (software) memory addresses into physical addresses (the ones the hardware sets in the address lines of the Avalon bus), and memory protection to prevent any process to write to memory sections without proper authorization, thus avoiding errant software execution.

- **Memory protection unit** *(MPU)* *(optional)*: This block is used when memory protection features are required but virtual memory management is not. It allows access permissions to the different regions in the memory map to be defined by software. In case a process attempts to perform an unauthorized memory access, an exception is generated.

- **JTAG debug module** *(optional)*: As shown in Figure 3.10, this block connects to the on-chip JTAG circuitry and to internal core signals. This allows the soft processor to be remotely accessed for debugging purposes. Some of the supported debugging tasks are downloading programs to memory, starting and stopping program execution, setting breakpoints and watchpoints, analyzing and editing registers and memory contents, and collecting real-time execution trace data. In this context, the advantage with regard to hard processors is that the debugging module can be used during the design and verification phase and removed for normal operation, thus releasing FPGA resources.
To ease the task of configuring the Nios-II architecture to fit the requirements of different applications, Altera provides three basic models from which designers can build their own core, depending on whether performance or complexity weighs more in their decisions. Nios-II/f (fast) is designed to maximize performance at the expense of FPGA resource usage. Nios-II/s (standard) offers a balanced trade-off between performance and resource usage. Finally, Nios-II/e (economy) optimizes resource usage at the expense of performance.

The similarities between the hardware architecture of Altera’s Nios-II and Xilinx MicroBlaze can be clearly noticed in Figure 3.8. Both are 32-bit RISC processors with Harvard architecture and include fixed and optional blocks, most of which are present in the two architectures, even if there may be some differences in the implementation details. Lattice’s LM32 is also a 32-bit RISC processor, but much simpler than the two former ones. For instance, it does not include an MMU block. It can be integrated with OSs such as μCLinux, uC/OS-II, and TOPPERS/JSP kernel (Lattice 2008).

The core processor is not the only element a soft processor consists of, but it is the most important one, since it has to ensure that any instruction in the ISA can be executed no matter what the configuration of the core is. In addition, the soft processor includes peripherals, memory resources, and the required interconnections. A large number of peripherals are or may be integrated in the soft processor architecture. They range from standard resources (GPIO, timers, counters, or UARTs) to complex, specialized blocks oriented to signal processing, networking, or biometrics, among other fields. Not only FPGA vendors provide peripherals to support their soft processors, but others are also available from third parties.

Communication of the core processor with peripherals and external circuits in the FPGA fabric is a key aspect in the architecture of soft processors. In this regard, there are significant differences among the three soft processors.
processors being analyzed. Nios-II has always used, from its very first versions to date, Altera’s proprietary Avalon bus. On the other hand, Xilinx initially used IBM’s CoreConnect bus, together with proprietary ones (such as local memory bus [LMB] and Xilinx CacheLink [XCL]), but the most current devices use ARM’s AXI interface. Lattice LM32 processor uses WishBone interfaces. A detailed analysis of the on-chip buses most widely used in FPPSoCs is made in Section 3.5.

At this point, readers may be afraid to realize the huge amount and diversity of concepts, terms, hardware and software alternatives, or design decisions one must face when dealing with soft processors. Fortunately, designers have at their disposal robust design environments as well as an ecosystem of design tools and IP cores that dramatically simplify the design process. The tools supporting the design of SoPCs are described in Section 6.3.

### 3.2.2 Open-Source Cores

In addition to proprietary cores, associated with certain FPGA architectures/vendors, there are also open-source soft processor cores available from other parties. Some examples are ARM’s Cortex-M1 and Cortex-M3, Freescale’s ColdFire V1, MIPS Technologies’ MP32, OpenRISC 1200 from OpenCores community, Aeroflex Gaisler’s LEON4, as well as implementations of many different well-known processors, such as the 8051, 80186 (88), and 68000. The main advantages of these solutions are that they are technology independent, low cost, based on well-known, proven architectures, and they are supported by a full set of tools and OSs.

The Cortex-M1 processor (ARM 2008), whose block diagram is shown in Figure 3.11a, was developed by ARM specifically targeting FPGAs. It has a 32-bit RISC architecture and, among other features, includes configurable instruction and data TCMs, interrupt controller, or configurable debug logic. The communication interface is ARM’s proprietary AMBA AHB-Lite 32-bit bus (described in Section 3.5.1.1). The core supports Altera, Microsemi, and Xilinx devices, and it can operate in a frequency range from 70 to 200 MHz, depending on the FPGA family.

The OpenRISC 1200 processor (OpenCores 2011) is based on the OpenRISC 1000 architecture, developed by OpenCores targeting the implementation of 32- and 64-bit processors. OpenRISC 1200, whose block diagram is shown in Figure 3.11b, is a 32-bit RISC processor with Harvard architecture. Among other features, it includes general-purpose registers, instructions and data caches, MMU, floating-point unit (FPU), MAC unit for the efficient implementation of signal processing functions, and exception/interrupt management units. The communication interface is WishBone (described in Section 3.5.4). It supports different OSs, such as Linux, RTEMS, FreeRTOS, and eCos.

LEON4 is a 32-bit processor based on the SPARC V8 architecture originated from European Space Agency’s project LEON. It is one of the most complex and flexible (configurable) open-source cores. It includes an ALU
with hardware multiply, divide, and MAC units, IEEE-754 FPU, MMU, and debug module with instruction and data trace buffer. It supports two levels of instruction and data caches and uses the AMBA 2.0 AHB bus (described in Section 3.5.1.1) as communication interface. From a software point of view, it supports Linux, eCos, RTEMS, Nucleus, VxWorks, and ThreadX.

Table 3.1 summarizes the performance of the different soft processors analyzed in this chapter. It should be noted that data have been extracted...
from information provided by vendors and, in some cases, it is not clear how this information has been obtained.

Since several soft processors can be instantiated in an FPGA design (to the extent that there are enough resources available), many diverse FPSoC solutions can be developed based on them, from single to multicore. These multicore systems may be based on the same or different soft processors, or their combination with hard processors, and support different OSs. Therefore, it is possible to design homogeneous or heterogeneous FPSoCs, with SMP or AMP architectures.

### 3.3 Hard Processors

Soft processors are a very suitable alternative for the development of FPSoCs, but when the highest possible performance is required, hard processors may be the only viable solution. Hard processors are commercial, usually proprietary, processors that are integrated with the FPGA fabric in the same chip, so they can be somehow considered as another type of specialized hardware blocks. The main difference with the stand-alone versions of the same processors is that hard ones are adapted to the architectures of the FPGA devices they are embedded in so that they can be connected to the FPGA fabric with minimum delay. However, very interestingly, from the point of view of software developers, there is no difference, for example, in terms of architecture or ISA.
There are obviously many advantages derived from the use of optimized, state-of-the-art processors. Their performance is similar to the corresponding ASIC implementations (and well known from these implementations); they have a wide variety of peripherals and memory management resources, are highly reliable, and have been carefully designed to provide a good performance/functionality/power consumption trade-off. Documentation is usually extensive and detailed, and they have whole development and support ecosystems provided by the vendors. There are also usually many reference designs available that designers can use as starting point to develop their own applications.

Hard processors have also some drawbacks. First, they are not scalable, because their fixed hardware structure cannot be modified. Second, since they are fine-tuned for each specific FPGA family, design portability may be limited. Finally, same as for stand-alone processors, obsolescence affects hard processors. This is a market segment where new devices with ever-enhanced features are continuously being released, and as a consequence, production of (and support for) relatively recent devices may be discontinued.

The first commercial FPSoCs including hard processors were proposed by Atmel and Triscend.* For instance, Atmel developed the AT94K Field Programmable System Level Integrated Circuit series (Atmel 2002), which combined a proprietary 8-bit RISC AVR processor (1 MIPS/MHz, up to 25 MHz) with reconfigurable logic based on its AT40K FPGA family. Triscend, on its side, commercialized the E5 series (Triscend 2000), including an 8032 microcontroller (8051/52 compatible, 10 MIPS at 40 MHz). In both cases, the reconfigurable part consisted of resources accounting for roughly 40,000 equivalent logic gates, and the peripherals of the microcontrollers consisted of just a small set of timers/counters, serial communication interfaces (SPI, UART), capture and compare units (capable of generating PWM signals), and interrupt controllers (capable of handling both internal and external interrupt sources). None of these devices is currently available in the market, although Atmel still produces AT40K FPGAs.

After only a few months, 32-bit processors entered the FPSoC market with the release of Altera’s Excalibur family (Altera 2002), followed by QuickLogic’s QuickMIPS ESP (QuickLogic 2001), Triscend’s A7 (Triscend 2001), and Xilinx’s Virtex-II Pro (Xilinx 2011b), Virtex-4 FX (Xilinx 2008), and Virtex-5 FXT (Xilinx 2010b). This was a big jump ahead in terms of processor architectures, available peripherals, and operating frequencies/performance.

Altera and Triscend already opted at this point to include ARM processors in their FPSoCs, whereas QuickLogic devices combined a MIPS32 4Kc processor from MIPS Technologies† with approximately 550,000 equivalent logic gates of Via-Link fabric (QuickLogic’s antifuse proprietary technology).

---

† Imagination Technologies acquired MIPS Technologies in 2013.
Xilinx Virtex-II Pro and Virtex-4 FX devices included up to two IBM PowerPC 405 cores and Virtex-5 FX up to two IBM PowerPC 440 cores. Only these three latter families are still in the market, although Xilinx recommends not to use them for new designs.

It took more than 5 years for a new FPSoC family (Microsemi’s SmartFusion, Figure 3.12) to be released, but since then there has been a tremendous evolution, with one common factor: All FPGA vendors opted for ARM architectures as the main processors for their FPSoC platforms. Microsemi’s SmartFusion and SmartFusion 2 (Microsemi 2013, 2016) families include an ARM Cortex-M3 32-bit RISC processor (Harvard architecture, up to 166 MHz, 1.25 DMIPS/MHz), with two 32 kB SRAM memory blocks, 512 kB of 64-bit nonvolatile memory, and 8 kB instruction cache. It provides different interfaces (all based on ARM’s proprietary AMBA bus, described in Section 3.5.1) for communication with specialized hardware blocks or custom user logic in the FPGA fabric, as well as many peripherals to support different communication standards (USB controller; SPI, I2C, and CAN blocks, multi-mode UARTs, or Triple-Speed Ethernet media access control). In addition, it includes an embedded trace macrocell block intended to ease system debug and setup.

Altera and Xilinx include ARM Cortex-A9 cores in some of their most recent FPGA families, such as Altera’s Arria 10 (Figure 3.13), Arria V,
The ARM Cortex-A9 is a 32-bit dual-core processor (2.5 DMIPS/MHz, up to 1.5 GHz). Dual-core architectures are particularly suitable for real-time operation, because one of the cores may run the OS and main application programs, whereas the other core is in charge of time-critical (real-time) functions. In both Altera and Xilinx devices, the processors and the FPGA fabric are supplied from separate power sources. If only the processor is to be used, it is possible to turn off power supply for the fabric, hence allowing power consumption to be reduced. In addition, the logic can be fully or partially configured from the processor at any time.*

The main features of the ARM Cortex-A9 dual-core processor are as follows:

- Ability to operate in single-processor, SMP dual-processor, or AMP dual-processor modes.
- Up to 256 kB of on-chip RAM and 64 kB of on-chip ROM.
- Each core has its own level 1 (L1) separate instruction and data caches, 32 kB each, and share 512 kB of level 2 (L2) cache.

* FPGA configuration is analyzed in Chapter 6.
Dynamic length pipeline (8–11 stages).

Eight-channel DMA controller supporting different data transfer types: memory to memory, memory to peripheral, peripheral to memory, and scatter–gather.

MMU.

Single- and double-precision FPU.

NEON media processing engine, which enhances FPU features by providing a quad-MAC and additional 64-bit and 128-bit register sets supporting single-instruction, multiple-data (SIMD) and vector floating-point instructions. NEON technology can accelerate multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis.

Available peripherals include interrupt controller, timers, GPIO, or 10/100/1000 tri-mode Ethernet Media Access Control, as well as USB 2.0, CAN, SPI, UART, and I2C interfaces.

Hard memory interfaces for DDR4, DDR3, DDR3L, DDR2, LPDDR2, flash (QSPI, NOR, and NAND), and SD/SDIO/MMC memories.

Connections with the FPGA fabric (distributed logic and specialized hardware blocks) through AXI interfaces (described in Section 3.5.1.3).

ARM CoreSight Program Trace Macrocell, which allows the instruction flow being executed to be accessed for debugging purposes (Sharma 2014).

At the time of writing of this book, the two most recently released FPSoC platforms are Altera’s Stratix 10 (Altera 2015d) and Xilinx’s Zynq UltraScale+ (Xilinx 2016b), both including an ARM Cortex-A53 quad-core processor. Most of the features of the ARM Cortex-A53 processor (Figure 3.14) are already present in the ARM Cortex-A9, but the former is smaller and has less power consumption. The cores in Stratix 10 and Zynq UltraScale+ families can operate up to 1.5 GHz, providing 2.3 DMIPS/MHz performance.

In addition, Zynq UltraScale+ devices include an ARM Cortex-R5 dual-core processor and an ARM Mali-400 MP2 GPU, as shown in Figure 3.14b, resulting in a heterogeneous multiprocessor SoC (MPSoC) hardware architecture. The ARM Cortex-R5 is a 32-bit dual-core real-time processor,* capable of operating at up to 600 MHz and providing 1.67 DMIPS/MHz performance. Cores can work in split (independent) or lock-step (parallel) modes.

---

* Cortex-A series includes “Application” processors and Cortex-R series “real-time” ones.
Embedded Processors in FPGA Architectures

FIGURE 3.14
Processing systems of (a) Altera’s Stratix 10 FPGAs and (b) Xilinx’s Zynq UltraScale+ MPSoCs.
Lock-step operation is intended for safety-critical applications requiring redundant systems.

The main features of each core are as follows:

- 32 kB L1 instruction and data caches and 128 kB TCM for highly deterministic or low-latency applications (real-time single-cycle access). All memories have ECC and/or parity protection.
- Interrupt controller.
- MPU.
- Single- and double-precision FPU.
- Embedded trace macrocell for connection to ARM CoreSight debugging system.
- AXI interfaces (described in Section 3.5.1.3).

The ARM Mali-400 GPU is a low-power graphics acceleration processor, capable of operating at up to 667 MHz. Its 2D vector graphics are based on OpenVG 1.1,* whereas 3D graphics are based on OpenGL ES 2.0.† It supports Full Scene Anti-Aliasing and Ericsson Texture Compression to reduce external memory bandwidth and is fully autonomous to operate in parallel with the ARM Cortex-A53 application processor. It consists of five main blocks:

1. **Geometry processor, in charge of the vertex processing stage of the graphics pipeline:** It generates lists of primitives and accelerates building of data structures for pixel processors.
2. **Pixel processors (two), which handle the rasterization and fragment processing stages of the graphics pipeline:** They produce the framebuffer results that screens display as final images.
3. **MMU:** Both the geometry processor and the pixel processors use MMUs for access checking and translation.
4. **L2 cache:** Geometry and pixels processors share a 64 kB L2 read-only cache.
5. **Power management unit, supporting power gating for all the other blocks.**

Some devices in the Zynq UltraScale+ family also include a video codec unit in the form of specialized hardware block (i.e., as part of the FPGA resources), supporting simultaneous encoding/decoding of video and audio streams. Its combination with the Mali-400 GPU results in a very suitable platform for multimedia applications.

---

* OpenVG 1.0 is a royalty-free, cross-platform API for hardware accelerated two-dimensional vector and raster graphics.
† OpenGL ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems.
All building blocks the processing system of Zynq UltraScale+ devices consist of are interconnected among themselves and with the FPGA fabric through AMBA AXI4 interfaces (described in Section 3.5.1.3).

Once hard and soft processors have been analyzed, it is important to emphasize that their features and performance (although extremely important*) are not the only ones to consider when addressing the design of FPSoCs. The resources available in the FPGA fabric (analyzed in Chapter 2) also play a fundamental role in this context.

Given the increasing complexity of FPSoC platforms, the availability of efficient software tools for design and verification tasks is also of paramount importance in the potential success of these platforms in the market. To realize how true this is, one has to just think about what it may take to debug a heterogeneous multicore FPSoC, where general-purpose and real-time OSs may have to interact (maybe also with some proprietary kernels) and share a whole bunch of hardware resources (memory and peripherals integrated in the processing system, implemented in the FPGA fabric, available there as specialized hardware blocks, or even implemented in external devices). Tools and methodologies for FPGA-based design are analyzed in Chapter 6, where special attention is paid to SoPC design tools (Section 6.3).

3.4 Other “Configurable” SoC Solutions

In previous sections, the most typical FPSoC solutions commercially available have been analyzed. They all have at least two common characteristics: the basic architecture, consisting of an FPGA and one or more embedded processors, and the fact that they target a wide range of application domains, that is, they are not focused on specific applications. This section analyzes other solutions with specific characteristics because either they do not follow the aforementioned basic architecture (some of them are not even based on FPGA and might have been excluded from this book, but are included to give readers a comprehensive view of configurable SoC architectures) or they target specific application domains.

3.4.1 Sensor Hubs

The integration in mobile devices (tablets, smartphones, wearables, and IoT) of multiple sensors enabling real-time context awareness (identification of user’s context) has contributed to the success of these devices. This is due to

*A clear conclusion deriving from the analyses in Sections 3.2 and 3.3 is that the main reason for the fast evolution of FPSoC platforms in recent years is related to the continuous development of more and more sophisticated SMP and AMP platforms.
the many services that can be offered based on the knowledge of data such as user state (e.g., sitting, walking, sleeping, or running), location, environmental conditions, or the ability to respond to voice commands. In order for the corresponding apps to work properly, it is necessary to have in place an always-on context aware monitoring and decision-making process involving data acquisition, storage and analysis, as well as a high computational power, because the necessary processing algorithms are usually very complex.

At first sight, one may think these are tasks that can be easily performed by traditional microcontroller- or DSP-based systems. However, in the case of mobile devices, power consumption from batteries becomes a fundamental concern, which requires specific solutions to tackle it. Real-time management of sensors implies a high power consumption if traditional processing platforms are used. This gave rise to a new paradigm, sensor hubs, which is very rapidly developing. Sensor hubs are coprocessing systems aimed at relieving a host processor from sensor management tasks, resulting in faster, more efficient, and less power-consuming (in the range of tens of microwatts) processing. They include the necessary hardware to detect changes in user’s context in real time. Only when the change of context requires host attention, it is notified and takes over the process.

QuickLogic specifically focuses on sensor hubs for mobile devices, offering two design platforms in this area, namely, EOS S3 Sensor Processing SoC (QuickLogic 2015) and Customer-Specific Standard Product (CSSP) (QuickLogic 2010).

EOS S3 is a sensor processing SoC platform intended to support a wide range of sensors in mobile devices, such as high-performance microphones, or environmental, inertial, or light sensors. Its basic architecture is shown in Figure 3.15. It consists of a multicore processor including a set of specialized hardware blocks and an FPGA fabric.

Control and processing tasks are executed in two processors, an ARM Cortex-M4F, including an FPU and up to 512 kB of SRAM memory, and a flexible fusion engine (FFE), which is a QuickLogic proprietary DSP-like (single-cycle MAC) VLIW processor. The ARM core is in charge of general-purpose processing tasks, and it hosts the OS, in case it is necessary to use one. The FFE processor is in charge of sensor data processing algorithms (such as voice triggering and recognition, motion-compensated heart rate monitoring, indoor navigation, pedestrian dead reckoning, or gesture detection). It supports in-system reconfiguration and includes a change detector targeting always-on context awareness applications.

A third processor, the Sensor Manager, is in charge of initializing, calibrating, and sampling front-end sensors (accelerometer, gyroscope, magnetometer, and pressure, ambient light, proximity, gesture, temperature, humidity, and heart rate sensors), as well as of data storage.

Data transfer among processors is carried out using multiple-packet FIFOS and DMA, whereas they connect with the sensors and the host processor mainly through SPI and I2C serial interfaces. Analog inputs connected to
12-bit sigma-delta ADCs are available for battery monitoring or for connecting low-speed analog peripherals.

Given the importance of audio in mobile devices, EOS S3 includes resources supporting always-listening voice applications. These include interfaces for direct connection of integrated interchip sound (I2S) and pulse-density modulation (PDM) microphones, a hardware PDM to pulse-code modulation (PCM) converter (which converts the output of low-cost PDM microphones to PCM for high-accuracy on-chip voice recognition without the need for using CODECs), and a hardware accelerator based on Sensory’s low power sound detector technology, in charge of detecting voice commands from low-level sound inputs. This block is capable of identifying if the sound coming from the microphone is actually voice, and only when this is the case, voice recognition tasks are carried out, providing significant energy savings.

Finally, the FPGA fabric allows the features of the FFE processor to be extended, the algorithms executed in either the ARM or the FFE processor to be accelerated, and user-defined functionalities to be added.

The CSSP platform was the predecessor of EOS S3 for the implementation of sensor hubs, but it can also support other applications related to connectivity and visualization in mobile devices. CSSP is not actually a family of devices, but a design approach, based on the use of configurable hardware platforms and a large portfolio of (mostly parameterizable) IP blocks, allowing the fast implementation of new products in the specific target application domains. The supporting hardware platforms are QuickLogic’s PolarPro and ArcticLink device families.

PolarPro is a family of simple devices with a few specialized hardware blocks such as RAM, FIFO, and (in the most complex devices) SPI and

**FIGURE 3.15**
EOS S3 block diagram.
I2C interfaces. ArcticLink is a family of specific-purpose FPGAs that includes (in addition to the serial communication interfaces mentioned in Section 2.4.5) FFE and sensor manager processors, similar to those available in EOS S3 devices, and processing blocks to improve visualization or reduce consumption in the displays. The types and number of functional blocks available in each device depend on the specific target application. Figure 3.16 shows possible solutions for the three main application domains of CSSP: connectivity, visualization, and sensor hub:

- **Connectivity applications** are those intended to facilitate the connection of the host processor with both internal resources and external devices such as keyboards, headphone jacks, or even computers. FPGAs with hard serial communication interfaces (e.g., PolarPro 3E or ArcticLink) offer a suitable support to these applications.

- **One of the most typical visualization problems in mobile devices** is the lack of compatibility between display and main CPU bus interfaces. To ease interface interconnection, some devices from the ArcticLink family include specialized hardware blocks serving as bridges between the most widely used display bus interfaces (namely, MIPI, RGB, and LVDS). For instance, the ArcticLink III VX5 family includes devices with MIPI input and LVDS output, RGB input and LVDS output, MIPI input and RGB output, or RGB input and MIPI output.

- The hard blocks **High Definition Visual Enhancement Engine (VEE HD+)** and **High Definition Display Power Optimizer (DPO HD+)** in ArcticLink devices are oriented to improve image visualization and reduce battery power consumption. VEE HD+ allows dynamic range, contrast, and color saturation in images to be optimized, improving image perception under different lighting conditions. DPO HD+ uses statistical data provided by VEE HD+ to adjust brightness, achieving significant energy savings (it should be noted that in these systems, displays are responsible for 30%–60% of the overall consumption).

- **CSSP supports sensor hub applications** through ArcticLink 3 S2 devices, which include FFE and Sensor Manager processors (similar to those available in EOS S3 devices) and a SPI interface for connection to the host applications processor.

In addition to their specialized hardware blocks, there is a large portfolio of soft IP blocks available for the devices supporting the CSSP platform, called **Proven System Blocks**. These include data storage, network connection, image processing, or security-related blocks, among others. Finally, both EOS S3 and CSSP have drivers available to integrate the devices with different OSs, such as Android, Linux, and Windows Mobile.
FIGURE 3.16
(a) Connectivity solution. (b) Visualization solution. (c) Sensor hub solution.
3.4.2 Customizable Processors

There are also non-FPGA-based configurable solutions offering designers a certain flexibility for the development of SoCs targeting specific applications. One such solution are customizable processors (Figure 3.17) (Cadence 2014; Synopsys 2015).

Customizable processors allow custom single- or multicore processors to be created from a basic core configuration and a given instruction set. Users can configure some of the resources of the processor to adapt its characteristics to the target application, as well as extend the instruction set by creating new instructions, for example, to accelerate critical functions.

Resource configuration includes the parameterization of some features of the core (instruction and data memory controllers, number of bits of internal buses, register structure, external communications interface, etc.), the possibility of adding or removing predefined components (such as multipliers, dividers, FPUs, DMA, GPIO, MAC units, interrupt controller, timers, or MMUs), or the possibility of adding new registers or user-defined components. This latter option is strongly linked to the ability to extend the instruction set, because most likely a new instruction will require some new hardware, and vice versa.

**FIGURE 3.17**
Customizable processors.
3.5 On-Chip Buses

One key factor to successfully develop embedded systems is to achieve an efficient communication between processors and their peripherals. Therefore, one of the major challenges of SoC technology is the design of the on-chip communication infrastructure, that is, the communication buses ensuring fast and secure exchange of information (either data or control signals), addressing issues such as speed, throughput, and latency. At the same time, it is very important (particularly when dealing with FPSoC platforms) that such functionality is available in the form of reusable IP cores, allowing both design costs and time to market to be reduced. Unfortunately, some IP core designers use different communication protocols and interfaces (even proprietary ones), complicating their integration and reuse, because of compatibility problems. In such cases, it is necessary to add glue logic to the designs. This creates problems related to degraded performance of the IP core and, in turn, of the whole SoC. To address these issues, over the years, some leading companies in the SoC market have proposed different on-chip bus architecture standards. The most popular ones are listed here:

- Advanced Microcontroller Bus Architecture (AMBA) from ARM (open standard)
- Avalon from Altera (open-source standard)
- CoreConnect from IBM (licensed, but available at no licensing or royalty cost for chip designers and core IP and tool developers)
- CoreFrame from PalmChip (licensed)
- Silicon Backplane from Sonics (licensed)
- STBus from STMicroelectronics (licensed)
- WishBone from OpenCores (open-source standard)

Most of these buses originated in association with certain processor architectures, for instance, AMBA (ARM processors), CoreConnect (PowerPC), or Avalon (Nios-II). Integration of a standard bus with its associated processor(s) is quite straightforward, resulting in modular systems with optimized and predictable behavior. Due to this, there is a trend, seen in the case of not only for chip vendors but also third-party IP companies, toward the use of technology-independent standard buses in library components, which ease design integration and verification.

In the FPGA market, AMBA has become the de facto dominating connectivity standard in industry for IP-based design because the leading vendors (Xilinx, Altera, Microsemi, QuickLogic) are clearly opting for embedding ARM processors (either Cortex-A or Cortex-M) within their chips. Other buses widely used in FPSoCs are Avalon and CoreConnect because of their...
association with the Nios-II and MicroBlaze soft processors, respectively. Wishbone is also used in some Lattice and OpenCores processors. These four buses are analyzed in detail in Sections 3.5.1 through 3.5.4.

3.5.1 AMBA

AMBA originated as the communication bus for ARM processor cores. It consists of a set of protocols included in five different specifications. The most widely used protocols in FPSoCs are Advanced eXtensible Interface (AXI3, AXI4, AXI4-Lite, AXI4-Stream) and Advanced High-performance Bus (AHB). Therefore, these are the ones analyzed in detail here, but at the end of the section, a table is included to provide a more general view of AMBA.

3.5.1.1 AHB

AMBA 2 specification, published in 1999, introduced AHB and Advanced Peripheral Bus (APB) protocols (ARM 1999). AMBA 2 uses by default a hierarchical bus architecture with at least one system (main, AHB) bus and secondary (peripheral, APB) buses connected to it through bridges. The performance and bandwidth of the system bus ensure the proper interconnection of high-performance, high clock frequency modules such as processors, on-chip memories, and DMA devices. Secondary buses are optimized to connect low-power or low-bandwidth peripherals, their complexity being, as a consequence, also low. Usually these peripherals use memory-mapped registers and are accessed under programmed control.

The structure of a SoC based on this specification is shown in Figure 3.18. The processor and high-bandwidth peripherals are interconnected through an AHB bus, whereas low-bandwidth peripherals are interconnected through an APB bus. The connection between these two buses is made through a bridge that translates AHB transfer commands into APB format and buffers all address, data, and control signals between both buses to accommodate their (usually different) operating frequencies. This structure allows the effect of slow modules in the communications of fast ones to be limited.

![FIGURE 3.18](image_url)  
SoC based on AHB and APB protocols.
In order to fulfill the requirements of high-bandwidth modules, AHB supports pipelined operation, burst transfers, and split transactions, with a configurable data bus width up to 128 bits. As shown in Figure 3.19, it has a master–slave structure with arbiter, based on multiplexed interconnections and four basic blocks: AHB master, AHB slave, AHB arbiter, and AHB decoder.

AHB masters are the only blocks that can launch a read or write operation, by generating the address to be accessed, the data to be transferred (in the case of write operations), and the required control signals. In an AHB bus, there may be more than one master (multimaster architecture), but only one of them can take over the bus at a time.

AHB slaves react to read or write requests and notify the master if the transfer was successfully completed, if there was an error in it, or if it could not be completed so that the master has to retry (e.g., in the case of split transactions).

The AHB arbiter is responsible to ensure only one AHB master takes over the bus (i.e., starts a data transfer) at a time. Therefore, it defines the bus access hierarchy, by means of a fixed arbitration protocol.

Finally, the AHB decoder is used for address decoding, generating the right slave selection signals. In an AHB bus, there is only one arbiter and one decoder.

Operation is as follows: All masters willing to start a transfer generate the corresponding address and control signals. The arbiter then decides which master signals are to be sent to all slaves through the corresponding MUXs, while the decoder selects the slave actually involved in the transfer through another MUX. In case there is an APB bus, it acts as
a slave of the corresponding bridge, which provides a second level of decoding for the APB slaves.

In FPGSoCs using AHB, the processor is a master; the DMA controller is usually a master too. On-chip memories, external memory controllers, and APB bridges are usually AHB slaves. Although any peripheral can be connected as an AHB slave, if there is an APB bus, slow peripherals would be connected to it.

### 3.5.1.2 Multilayer AHB

AMBA 3 specification (ARM 2004a), published in 2003, introduces the multilayer AHB interconnection scheme, based on an interconnection matrix that allows multiple parallel connections between masters and slaves to be established. This provides increased flexibility, higher bandwidth, the possibility of associating the same slave to several masters, and reduced complexity, because arbitration tasks are limited to the cases when several masters want to access the same slave at the same time.

The simplest multilayer AHB structure is shown in Figure 3.20, where each master has its own AHB layer (i.e., there is only one master per layer). The decoder associated with each layer determines the slave involved in the transfer. If two masters request access to the same slave at the same time, the arbiter associated with the slave decides which master has the higher priority. The input stages of the interconnection matrix (one per layer) store the addresses and control signals corresponding to the pending transfers so that can be carried out later.

The number of input and output ports of the interconnection matrix can be adapted to the requirements of different applications. In this way, it is possible to build structures more complex than the one in Figure 3.20. For instance,

---

**FIGURE 3.20**
Multilayer interconnect topology.
it is possible to have several masters in the same layer, define local slaves (connected to just one layer), or group a set of slaves so that the interconnection matrix treats them as a single one. This is useful, for instance, to combine low-bandwidth slaves.

An example of FPGSoC that uses AHB/APB buses is the Microsemi SmartFusion2 SoC family (Microsemi 2013). As shown in Figure 3.21, it includes an ARM Cortex-M3 core and a set of peripherals organized in 10 masters (MM), 7 direct slaves (MS), and a large number of secondary slaves, connected through an AHB to AHB bridge and two APB bridges (APB_0 and APB_1). The AHB bus matrix is multilayer.

### 3.5.1.3 AXI

ARM introduced in AMBA 3 specification a new architecture, namely, AXI or, more precisely, AXI3 (ARM 2004b). The architecture was provided with additional functionalities in AMBA 4, resulting in AXI4 (ARM 2011). AXI provides a very efficient solution for communicating with high-frequency peripherals, as well as for multifrequency systems (i.e., systems with multiple clock domains).

AXI is currently a de facto standard for on-chip busing. A proof of its success is that some 35 leading companies (including OEM, EDA, and chip designers—FPGA vendors among them) cooperate in its development. As a result, AXI provides a communication interface and architecture suitable for SoC implementation in either ASICs or FPGAs.

AMBA 3 and AMBA 4 define four different versions of the protocol, namely, AXI3, AXI4, AXI4-Lite, and AXI4-Stream. Both AXI3 and AXI4 are
very robust, high-performance, memory-mapped solutions.* AXI4-Lite is a very reduced version of AXI4, intended to support access to control registers and low-performance peripherals. AXI4-Stream is intended to support high-speed streaming applications, where data access does not require addressing.

As shown in Figure 3.22, AXI architecture is conceptually similar to that of AHB in that both use master–slave configurations, where data transfers are launched by masters and there are interconnect components to connect masters to slaves.

The main difference is that AXI uses a point-to-point channel architecture, where address and control signals, read data, and write data use independent channels. This allows simultaneous, bidirectional data transfers between a master and a slave to be carried out, using handshake signals. A direct implication of this feature is that it eases the implementation of low-cost DMA systems.

AXI defines a single connection interface either to connect a master or a slave to the interconnect component or to directly connect a master to a slave. This interface has five different channels: read address channel, read data channel, write address channel, write data channel, and write response channel. Figure 3.23 shows read and write transactions in AXI.

Address and control information is sent through either the read or the write address channels. In read operations, the slave sends the master both data and a read response through the read data channel. The read response notifies the master that the read operation has been completed. The protocol includes an overlapping read burst feature, so the master may send a new read address before the slave has completed the current transaction. In this way, the slave can start preparing data for the new transaction while completing the current one, thus speeding up the read process. In write operations, the master sends data through the write data channel, and the slave

* Memory-mapped protocols refer to those where each data transfer accesses a certain address within a memory space (map).
Embedded Processors in FPGA Architectures

replies with a completion signal through the write response channel. Write data are buffered, so the master can start a new transfer before the slave notifies the completion of the current one. Read and write data bus widths are configurable from 8 to 1024 bits. All data transfers in AXI (except AXI4-Lite) are based on variable-length bursts, up to 16 transfers in AXI3 and up to 256 in AXI4. Only the starting address of the burst needs to be provided to start the transfer.

The interconnect component in Figure 3.22 is more versatile than the interconnection matrix in AHB. It is a component with more than one AMBA interface, in charge of connecting one or more masters to one or more slaves. In addition, it allows a set of masters or slaves to be grouped together, so they are seen as a single master or slave.

In order to adapt the balance between performance and complexity to different application requirements, the interconnect component can be configured in several modes. The most usual ones are shared address and data buses, shared address buses and multiple data buses, and multilayer, with multiple address and data buses. For instance, in systems requiring much higher bandwidth for data than for addresses, it is possible to share the address bus among different interfaces while having an independent data bus for each interface. In this way, data can be transferred in parallel at the same time as address channels are simplified.

FIGURE 3.23
Read (a) and write (b) transactions in AXI protocol.
Other interesting features of AXI are as follows:

- It supports pipeline stages (register slices in ARM’s terminology) in all channels, so different throughput/latency trade-offs can be achieved depending on the number of stages. This is feasible because all channels are independent of each other and send information in only one direction.

- Each master–slave pair can operate at a different frequency, thus simplifying the implementation of multifrequency systems.

- It supports out-of-order transaction completion. For instance, if a master starts a transaction with a slow peripheral and later another one with a fast peripheral, it does not need to wait for the former to be completed before attending the latter (unless completing the transactions in a given order is a requirement of the application). In this way, the negative influence of dead times caused by slow peripherals is reduced. Complex peripherals can also take advantage of this feature to send their data out of order (some complex peripherals may generate different data with different latencies). Out-of-order transactions are supported in AXI by ID tags. The master assigns the same ID tag to all transactions that need to be completed on order and different ID tags to those not requiring a given order of completion.

We are just intending here to highlight some of the most significant features of AXI, but it is really a complex protocol because of its versatility and high degree of configurability. It includes many other features, such as unaligned data transfers, data upsizing and downsizing, different burst types, system cache, privileged and secure accesses, semaphore-type operations to enable exclusive accesses, and error support.

Today, the vast majority of FPSoCs use this type of interface, and vendors include a large variety of IP blocks based on it, which can be easily connected to create highly modular systems. In most cases, when including AXI-based IPs in a design, the interconnect logic is automatically generated and the designer usually just needs to define some configuration parameters.

The most important conclusion that can be extracted from the use of this solution is that it enables software developers to implement SoCs without the need for deep knowledge of FPGA technology, but mainly concentrating on programming tasks.

As an example, Xilinx adopted AXI as a communication interface for the IP cores in its FPGA families Spartan-6, Virtex-6, UltraScale, 7 series, and Zynq-7000 All Programmable SoC (Sundaramoorthy et al. 2010; Singh and Dao 2013; Xilinx 2015a). The portfolio of AXI-compliant IP cores includes a large number of peripherals widely used in SoC
design, such as processors, timers, UARTs, memory controllers, Ethernet controllers, video controllers, and PCIe. In addition, a set of resources known as Infrastructure IP are also available to help in assembling the whole FPSoC. They provide features such as routing, transforming, and data checking.

Examples of such blocks are as follows:

- **AXI Interconnect IP**, to connect memory-mapped masters and slaves. It performs the tasks associated with the interconnect component by combining a set of IP cores (Figure 3.24):
  
  As commented earlier, AXI does not define the structure of the interconnect component, but it can be configured in multiple ways. The AXI Interconnect IP core supports the use models shown in Figure 3.25, which highlights the versatility and power of AXI for the implementation of FPSoCs.

- **AXI Crossbar**, to connect AXI memory-mapped peripherals.

- **AXI Data Width Converter**, to resize the datapath when master and slave use different data widths.

- **AXI Clock Converter**, to connect masters and slaves operating in different clock domains.

- **AXI Protocol Converter**, to connect an AXI3, AXI4, or AXI4-Lite master to a slave that uses a different protocol (e.g., AXI4 to AXI4-Lite or AXI4 to AXI3).

- **AXI Data FIFO**, to connect a master to a slave through FIFO buffers (it affects read and write channels).

- **AXI Register Slice**, to connect a master to a slave through a set of pipeline stages. In most cases, this is intended to reduce critical path delay.

- **AXI Performance Monitors and Protocol Checkers**, to test and debug AXI transactions.

![FIGURE 3.24](image)

Block diagram of the Xilinx’s AXI Interconnect IP core.
In order for readers to have easy access to the most significant information regarding the different variations of AMBA, their main features are summarized in Table 3.2.

### 3.5.2 Avalon

Avalon is the solution provided by Altera to support FPSoC design based on the Nios-II soft processor. The original specification dates back to 2002, and a slightly modified version can be found in Altera (2003).

Avalon basically defines a master–slave structure with arbiter, which supports simultaneous data transfers among multiple master–slave pairs. When multiple

![Avalon Diagram](image-url)
<table>
<thead>
<tr>
<th>Year</th>
<th>Spec.</th>
<th>Protocol</th>
<th>Aim and Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>AMBA 2</td>
<td>AHB</td>
<td>Supports high-bandwidth system modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Main system bus in microcontroller usage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Some features are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 32-bit address width and 8- to 128-bit data width</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Single shared address bus and separate read and write data buses</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Default hierarchical bus topology support</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Supports multiple bus masters</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Burst transfers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Split transactions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Pipelined operation (fixed pipeline between address/control and data phases)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Single-cycle bus master handover</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Single-clock edge operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Non-tri-state implementation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Single frequency system</td>
</tr>
<tr>
<td>APB</td>
<td></td>
<td></td>
<td>Simple, low-power interface to support low-bandwidth peripherals</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Some features are</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Local secondary bus encapsulated as a single AHB slave device</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 32-bit address width and 32-bit data width</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Simple interface</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Latched address and control</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Minimal gate count for peripherals</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Burst transfers not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Unpipelined</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• All signal transitions are only related to the rising edge of the clock</td>
</tr>
</tbody>
</table>

ASB   Obsolete
### TABLE 3.2 (Continued)

**Specifications and Protocols of the AMBA Communication Bus**

<table>
<thead>
<tr>
<th>Year</th>
<th>Spec.</th>
<th>Protocol</th>
<th>Aim and Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>AMBA 3</td>
<td>AXI (AXI3)</td>
<td>Intended for high-performance memory-mapped requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Key features:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 32-bit address width and 8- to 1024-bit data width</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Five separate channels: read address, write address, read data, write data, and write response</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Default bus matrix topology support</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Simultaneous read and write transactions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Support for unaligned data transfers using byte strobes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Burst-based transactions with only start address issued</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Fixed-burst mode for memory-mapped I/O peripherals</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Ability to issue multiple outstanding addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Out-of-order transaction completion</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Pipelined interconnect for high-speed operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Register slices can be applied across any channel</td>
</tr>
<tr>
<td></td>
<td>AHB-Lite</td>
<td></td>
<td>The main differences with regard to AHB are that it does not support multiple bus masters and extends data width up to 1024 bits</td>
</tr>
<tr>
<td></td>
<td>APB</td>
<td></td>
<td>Includes two new features with regard to AMBA 2 specification, namely, wait states and error reporting</td>
</tr>
<tr>
<td></td>
<td>ATB</td>
<td></td>
<td><em>Advanced Trace Bus</em>: adds a data diagnostic interface to the AMBA specification for debugging purposes</td>
</tr>
</tbody>
</table>

(Continued)
**TABLE 3.2 (Continued)**

Specifications and Protocols of the AMBA Communication Bus

<table>
<thead>
<tr>
<th>Year</th>
<th>Spec.</th>
<th>Protocol</th>
<th>Aim and Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>AMBA4</td>
<td>ACE</td>
<td>AXI Coherency Extensions: extends the AXI4 protocol and provides support for hardware-coherent caches. Enables correctness to be maintained when sharing data across caches</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACE-Lite</td>
<td>Small subset of ACE signals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AXI4</td>
<td>The main difference with regard to AXI3 is that it allows up to 256 beats of data per burst instead of just 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AXI4-Lite</td>
<td>It supports Quality of Service signaling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AXI4-Stream</td>
<td>A subset of AXI4 intended for simple, low-throughput memory-mapped communications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Key features:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Burst length of one for all transactions</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 32- or 64-bit data bus</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exclusive accesses not supported</td>
<td></td>
</tr>
<tr>
<td>2013</td>
<td>AMBA5</td>
<td>CHI</td>
<td>Coherent Hub Interface: it defines the interconnection interface for fully coherent processors and dynamic memory controllers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Used in networks and serves</td>
</tr>
<tr>
<td></td>
<td></td>
<td>APB</td>
<td>Includes two new functionalities with regard to AMBA 3 specification, namely, transaction protection and sparse data transfer</td>
</tr>
</tbody>
</table>
masters want to access the same slave, the arbitration logic defines the access priority and generates the control signals required to ensure all requested transactions are eventually completed. Figure 3.26 shows the block diagram of a sample FPSoC including a set of peripherals connected through an Avalon Bus Module.

The Avalon Bus Module includes all address, data, and control signals, as well as arbitration logic, required to connect the peripherals and build up the FPSoC. Its functionality includes address decoding for peripheral selection, wait-state generation to accommodate slow peripherals that cannot provide responses within a single clock cycle, identification and prioritization of interrupts generated by slave peripherals, or dynamic bus sizing to allow peripherals with different data widths to be connected. The original Avalon specification supports 8-, 16-, and 32-bit data.

Avalon uses separate ports for address, data, and control signals. In this way, the design of the peripherals is simplified, because there is no need for decoding each bus cycle to distinguish addresses from data or to disable outputs.

Although it is mainly oriented to memory-mapped connections, where each master–slave pair exchanges a single datum per bus transfer, the original Avalon specification also includes streaming peripherals and latency-aware peripherals modes (included in the Avalon Bus Module), oriented to support high-bandwidth peripherals. The first one eases transactions between streaming master and streaming slave to perform successive data transfers, which is particularly interesting for DMA transfers. The second one allows bandwidth usage to be optimized when accessing synchronous peripherals that require an initial latency to generate the first datum, but after that are capable of generating a new one each clock cycle (such as in the

---

**FIGURE 3.26**
Sample FPSoC based on Altera’s Avalon bus.
case of digital filters). In this mode, the master can execute a read request to the peripheral, then move to other tasks, and resume the read operation later.

As the demand for higher bandwidth and throughput was growing in many application domains, Avalon and the Nios-II architecture evolved to cope with it. The current Avalon specification (Altera 2015a) defines seven different interfaces:

1. Avalon Memory Mapped Interface (Avalon-MM), oriented to the connection of memory-mapped master–slave peripherals. It provides different operation modes supporting both simple peripherals requiring a fixed number of bus cycles to perform read or write transfers and much more complex ones, for example, with pipelining or burst capabilities. With regard to the original specification, maximum data width increases from 32 to 1024 bits.

   Like AMBA and many other memory-mapped buses, Avalon provides generic control and handshake signals to indicate the direction (read or write), start, end, successful completion, or error of each data transfer. Examples of such signals are “read,” “write,” or “response” in Figure 3.27. There are also specific signals required in advanced modes, such as arbitration signals in multimaster systems, wait signals to notify the master the slave cannot provide an immediate response to the request (“wait_request” in Figure 3.27), data valid signals (typical in pipelined peripherals to notify the master that there are valid data in the data bus, “read_data_valid” in Figure 3.27), or control signals for burst transfers.

2. Avalon Streaming Interface (Avalon-ST, Figure 3.28), oriented to peripherals performing high-bandwidth, low-latency, unidirectional point-to-point transfers. The simplest version supports single stream

![Figure 3.27](image-url)  
Typical read and write transfers of the Avalon-MM interface.
of data, which only requires the signals “data” and “valid” to be used and, optionally, “channel” and “error.” The sink interface samples data only if “valid” is active (i.e., there are valid data in “data”). The signal “channel” indicates the number of the channel, and “error” is a bit mask stating the error conditions considered in the data transfer (e.g., bit 0 and bit 1 may flag CRC and overflow errors, respectively).

Avalon-ST also allows interfaces supporting backpressure to be implemented. In this case, the source interface can only send data to the sink when this is ready to accept them (the signal “ready” is active). This is a usual technique to prevent data loss, for example, when the FIFO at the sink is full.

Finally, Avalon-ST supports burst and packet transfers. In packet-based transfers, “startofpacket” and “endofpacket” identify the first and last valid bus cycles of the packet. The signal “empty” identifies empty symbols in the packet, in the case of variable-length packets.

3. Avalon Conduit Interface, which allows data transfer signals (input, output, or bidirectional) to be created when they do not fit in any other types of Avalon interface. These are mainly used to design interfaces with external (off-chip) devices. Several conduits can be connected if they use the same type of signals, of the same width, and within the same clock domain.

4. Avalon Tri-State Conduit Interface (Avalon-TC), oriented to the design of controllers for external devices sharing resources such as address or data buses, or control signals in the terminals of the FPGA chip. Signal multiplexing is widely used to access multiple external devices minimizing the number of terminals required. In this case, the access to the shared terminals is based on tri-state signals. Avalon-TC includes all control and arbitration logic to identify multiplexed signals and give bus control to the right peripheral at any moment.

5. Avalon Interrupt Interface, which is in charge of managing interrupts generated by interrupt senders (slave peripherals) and notify them to the corresponding interrupt receivers (masters).
6. Avalon Reset Interface, which resets the internal logic of an interface or peripheral, forcing it to a user-defined safe state.

7. Avalon Clock Interface, which defines the clock signal(s) used by a peripheral. A peripheral may have clock input (clock sink), clock output (clock source), or both (for instance, in the case of PLLs). All other synchronous interfaces a peripheral may use (MM, ST, Conduit, TC, Interrupt, or Reset) are associated with a clock source acting as synchronization reference.

An FPSoC based on the Nios-II processor and Avalon may include multiple different interfaces or multiple instances of the same interface. Actually, a single component within the FPSoC may use any number and type of interfaces, as shown in Figure 3.29.

To ease the design and verification of Avalon-based FPSoCs, Altera provides the system integration tool Qsys (Altera 2015b), which automatically generates the suitable interconnect fabric (address/data bus connections, bus width matching logic, address decoder logic, arbitration logic) to connect a large number of IP cores available in its design libraries. Actually,
Qsys also eases the design of systems using both Avalon and AXI and automatically generates bridges to connect components using different buses (Altera 2013).

### 3.5.3 CoreConnect

CoreConnect is an on-chip interconnection architecture proposed by IBM in the 1990s. Although the current strong trend to use ARM cores in the most current FPGA devices points to the supremacy of AMBA-based solutions, CoreConnect is briefly analyzed here because Xilinx uses it for the MicroBlaze (soft) and PowerPC (hard) embedded processors.

CoreConnect consists of three different buses, intended to accommodate memory-mapped or DMA peripherals of different performance levels (IBM 1999; Bergamaschi and Lee 2000):

1. Processor Local Bus (PLB), a system bus to serve the processor and connect high-bandwidth peripherals (such as on-chip memories or DMA controllers).
2. On-Chip Peripheral Bus (OPB), a secondary bus to connect low-bandwidth peripherals and reduce traffic in PLB.
3. Device Control Register (DCR), oriented to provide a channel to configure the control registers of the different peripherals from the processor and mainly used to initialize them.

The block diagram of the CoreConnect bus architecture is shown in Figure 3.30, where structural similarities with AMBA 2 (Figure 3.18) may be noticed. Same as AMBA 2, CoreConnect uses two buses, PLB and OPB, with different performance levels, interconnected through bridges.

**FIGURE 3.30**
Sample FPSoC using CoreConnect bus architecture.
Both PLB and OPB use independent channels for addresses, read data, and write data. This enables simultaneous bidirectional transfers. They also support a multimaster structure with arbiter, where bus control is taken over by one master at a time.

PLB includes functionalities to improve transfer speed and safety, such as fixed- or variable-length burst transfers, line transfers, address pipelining (allowing a new read or write request to be overlapped with the one currently being serviced), master-driven atomic operation, split transactions, or slave error reporting, among others.

PLB-to-OPB bridges allow PLB masters to access OPB peripherals, therefore acting as OPB masters and PLB slaves. Bridges support dynamic bus sizing (same as the buses themselves), line transfers, burst transfers, and DMA transfers to/from OPB masters.

Former Xilinx Virtex-II Pro and Virtex-4 families include embedded PowerPC 405 hard processors (Xilinx 2010a), whereas PowerPC 440 processors are included in Virtex-5 devices (Xilinx 2010b). In all cases, CoreConnect is used as communication interface. Specifically, PLB buses are used for data transfers and DCR for initializing the peripherals as well as for system verification purposes.

Although the most recent versions of the MicroBlaze soft processor (from 2013.1 on) use as main interconnection interfaces AMBA 4 (AXI4 and ACE) and Xilinx proprietary bus LMB, optionally, they can implement OPB.

## 3.5.4 WishBone

Wishbone Interconnection for Portable IP Cores (usually referred to just as Wishbone) is a communication interface developed by Silicore in 1999 and maintained since 2002 by OpenCores. Like the other interfaces described so far, Wishbone is based on a master–slave architecture, but, unlike them, it defines just one bus type, a high-speed bus. Systems requiring connections to both high-performance (i.e., high-speed, low-latency) and low-performance (i.e., low-speed, high-latency) peripherals may use two separate Wishbone interfaces without the need for using bridges.

The general Wishbone architecture is shown in Figure 3.31. It includes two basic blocks, namely, SYSCON (in charge of generating clock and reset signals) and INTERCON (the one containing the interconnections). It supports four different interconnection topologies, some of them with multimaster capabilities:

- Point to point, which connects a single master to a single slave.
- Data flow, used to implement pipelined systems. In this topology, each pipeline stage has a master interface and a slave interface.
- Shared bus, which connects two or more masters with one or more slaves, but only allows one single transaction to take place at a time.
- Crossbar switch, which allows two or more masters to be simultaneously connected to two or more slaves; that is, it has several connection channels.

Shared bus and crossbar switch topologies require arbitration to define how and when each master accesses the slaves. However, arbiters are not defined in the Wishbone specification, so they have to be user defined.

According to Figure 3.31, Wishbone interfaces have independent address (ADR, 64-bit) and data (DAT, 8-/16-/32- or 64-bit) buses, as well as a set of handshake signals (selection [SEL], strobe [STB], acknowledge [ACK], error [ERR], retry [RTY], and cycle [CYC]) ensuring correct transmission of information and allowing data transfer rate to be adjusted for every bus cycle (all Wishbone bus cycles run at the speed of the slowest interface).

In addition to the signals defined in its specification, Wishbone supports user-defined ones in the form of “tags” (TAGN in Figure 3.31). These may be used for appending information to an address bus, a data bus, or a bus cycle. They are especially helpful to identify information such as data transfers, parity or error correction bits, interrupt vectors, or cache control operations.
Wishbone supports three basic data transfer modes:

1. Single read/write, used in single-data transfers.
2. Block read/write, used in burst transfers.
3. Read–modify–write, which allows data to be both read and written in a given memory location in the same bus cycle. During the first half of the cycle, a single read data transfer is performed, whereas a write data transfer is performed during the second half. The CYC_O signal (Figure 3.31) remains asserted during both halves of the cycle. This transfer mode is used in multiprocessor or multitask systems where different software processes share resources using semaphores to indicate whether a given resource is available or not at a given moment.

Wishbone is used in Lattice’s LM8 and LM32, as well as in OpenCores’ OpenRISC1200 soft processors, described in Sections 3.2.1 and 3.2.2, respectively.

References

Altera. 2015a. Avalon interface specifications. MNLAVABUSREF 2015.03.04.
Altera. 2015b. Quartus prime standard edition handbook. QPSV1 2015.05.04.
Altera. 2015d. Stratix 10 device overview data sheet. S10-OVERVIEW.
ARM. 1999. AMBA specification (rev 2.0) datasheet. IHI 0011A.
ARM. 2004a. Multilayer AHB overview datasheet. DVI 0045B.
ARM. 2004b. AMBA AXI protocol specification (v1.0) datasheet. IHI 0022B.
ARM. 2011. AMBA AXI and ACE protocol specification datasheet. IHI 0022D.
Atmel. 2002. AT94K series field programmable system level integrated circuit data sheet. 1138F-FPSLI-06/02.
Microsemi. 2013. SmartFusion2 microcontroller subsystem user guide.
OpenCores. 2011. OpenRISC 1200 IP core specification (v0.11).
QuickLogic. 2001. QL901M QuickMIPS data sheet.
QuickLogic. 2015. QuickLogic EOS S3 sensor processing SoC platform brief. Datasheet.
Synopsys. 2015. DesignWare ARC HS34 processor datasheet.
Walls, C. 2014. Selecting an operating system for embedded applications. Mentor
Graphics white paper TECH112110-w.
Xilinx. 2008. Virtex-4 FPGA user guide UG070 (v2.6).
Xilinx. 2010b. Embedded processor block in Virtex-5 FPGAs reference guide UG200
(v1.8).
Xilinx. 2011b. Virtex-II Pro and Virtex-II Pro X platform FPGAs: Complete data sheet
DS083 (v5.0).
Xilinx. 2014. Zynq-7000 all programmable SoC technical reference manual UG585
(v1.7).
Xilinx. 2015b. Xilinx collaborates with TSMC on 7nm for fourth consecutive generation
of all programmable technology leadership and multi-node scaling advantage.
Available at http://press.xilinx.com/2015-05-28-Xilinx-Collaborates-with-TSMC-
Xilinx. 2016b. Zynq UltraScale+ MPSoC overview data sheet DS891 (v1.1).