The design of an ASIP is a challenging task due to the large number of design options. The competing design decisions such as flexibility, performance, and energy consumption need to be weighted against each other to reach the optimal point in the entire design space. Moreover, the increasing software complexity of today’s SoCs requires a shift from traditional assembly programming to high-level languages to boost the designer’s productivity. As a result, processor designers demand an increasing support from the design automation tools to explore the design space and rightly balance the flexibility vs. performance trade-off.

Section 2.1 first presents the four major phases in an ASIP design. Afterward, Section 2.2 elaborates on the benefits and issues of compiler-in-the-loop architecture exploration. Finally, Section 2.3 presents prominent ASIP design methodologies. A survey of different ASIP design environments is given in [171].

2.1 ASIP Design Phases

The design of an ASIP is a highly complex task requiring diverse skills in different areas. The design process can be separated into four interrelated phases (Fig. 2.1):

![ASIP design phases](image)

**Architecture exploration**: The target application is mapped onto a processor architecture in an iterative process that is repeated until a best fit between architecture and application is obtained. According to Amdahl’s law [88], the application’s hot spots need to be optimized to achieve high performance.
improvements, and hence constitute promising candidates for dedicated hardware support and custom instructions. In order to identify those hot spots, profiling tools such as in [148, 203] are employed. Based on this hardware/software partitioning the instruction-set architecture (ISA) is defined in a second step. Afterward, the micro-architecture needs to be designed that implements the ISA. The whole process requires an architecture-specific set of software development tools (compiler, assembler, linker, simulator, and profiler). Unfortunately, every change to the architecture specification requires a complete new set of software development tools.

Architecture implementation: The specified processor is converted into a synthesizable hardware description language (HDL) model. For this purpose, languages such as VHDL [121] or Verilog [120] are employed. This model can then be further used for a standard synthesis flow (e.g., design compiler [250]). With this additional transformation, quite naturally, considerable consistency problems can arise between the architecture specification, the software development tools, and the hardware implementation.

Software application design: Software designers need a set of production-quality software development tools for efficient application design. However, the demands of the software application designer and the hardware processor designer place different requirements on software development tools. For example, the processor designer needs a cycle/phase-accurate simulator for hardware–software partitioning and profiling, which is very accurate, but inevitably slow. The application designer in contrast demands more simulation speed than accuracy. At this point, the complete set of software development tools is usually re-implemented by hand, which leads to consistency problems.

System integration and verification: The designed ASIP must be integrated into a system simulation environment of the entire SoC for verification. Since the interaction of all SoC components may have an impact on the processor performance, this provides more accurate results as compared to an instruction-set simulator. However, in order to integrate the software simulator, co-simulation interfaces must be developed. Again, manual modifications of the interfaces are required with each change of the architecture.

In traditional ASIP design, these phases are processed sequentially and are assigned to different design groups each with expert knowledge in the respective field. Design automation – if available at all – is mostly limited to the individual phases. Moreover, results in one phase may impose modifications in other phases. As a result, the complexity of design team interactions and communications necessary to successfully undertake a SoC-based design is a significant time-consuming factor. What makes this even more challenging is the large number of design alternatives that need to be weighted against each other. Consequently, the designer’s productivity becomes the vital factor for successful products due to the complexity and tight time-to-market constraints. As a result, there is a strong interest in comprehensive design methodologies for efficient embedded processor optimization and exploration.
Much of the functionality in a SoC is implemented in software due to a number of reasons: the flexibility of software offers wide design reuse (to reduce NRE costs) and compatibility across applications. It is conjectured that the amount of software in embedded systems roughly doubles every 2 years [85]. As a result, a rapidly increasing amount of software has to be validated and/or developed. This involves not only essential hardware drivers but also complete operating systems. Furthermore, new applications, exploiting the new hardware capabilities, need to be developed before the end products based on the SoC can be sold.

Compilers are among the most widespread software tools, used for decades on desktop computer. For embedded processors, however, the use of compilers is traditionally less common. Many designers still prefer assembly languages due to efficiency reasons. Considering the increasing complexity of applications and today’s short time-to-market windows, assembly programming is no longer feasible due to the huge programming effort, portability, and maintainability. Obviously, such requirements can be much better met by using high-level language (HLL) compilers. In the context of embedded systems, the C programming language [45] is widely used. It is a well-tried programming language that allows a very low-level programming style at a stretch. Additionally, this enables a broad design reuse since there already exists a large amount of industry standards and legacy code in C. Unfortunately, designing a compiler is a complex task that demands expert knowledge and a large amount of human resources. As a result, compilers are often not available for newly designed processors. Clearly, this increases the probability of designing a strong compiler-unfriendly architecture, which leads to an inefficient application implementation in the end. In fact, many in-house ASIP design projects suffer from the late development of the compiler. Compiler Designers often have severe difficulties ensuring good code quality due to instruction-sets that have primarily been designed from a hardware designer’s perspective. On the other hand, a compiler-friendly instruction-set and architecture might not be entirely suitable to support the hardware designer’s effort meeting constraints such as area and power consumption. Therefore, compiler-in-the-loop architecture exploration is crucial to avoid a compiler and architecture mismatch right from the beginning and to ensure an efficient application design for successful products.

The inherently application-specific nature of embedded processors leads to a wide variety of embedded processor architectures. Understandably, developing the software tools, in particular the compiler, for each processor is costly and extremely time-consuming. Therefore, retargetable C compilers have found significant use in ASIP design in the past years since they can be quickly adapted to varying processor configurations. This is also a result of the increasing tool support for automatically retargeting a C compiler based on formalized processor descriptions [224].

In compiler-in-the-loop architecture exploration the compiler plays a key role to obtain exploration results. Due to the ambiguity of the transformation of C applications to assembly code, it is possible to quickly evaluate fundamental architectural changes with minimal modifications of the compiler [194]. In this way, designers
can meaningfully and rapidly explore the design space by accurately tracking the impact of changes to the instruction-set, instruction latencies, register file size, etc. This is an important piece in the puzzle to better understand the mutual dependencies between micro-architecture design, the respective instruction-set, compilers, and the achieved code quality. What is most important in this context is the specification of the compiler’s code selector. It basically describes the mapping of the source code to an equivalent sequence of assembly instruction and hence significantly affects the final ISA definition (i.e., the software/hardware partitioning). However, the success of compiler-aided architecture exploration strongly depends on a flexible C compiler backend that is generated from the processor description.

Even though retargetable compilers have found significant use in ASIP design in the past years, they are still hampered by their limited code quality as compared to handwritten compilers or assembly code. This is actually no surprise, since higher compiler flexibility comes at the expense of a lower amount of target-specific code optimizations. Since such compilers can only make few assumptions about the target machine, it is, understandably, much easier to support machine-independent optimizations rather than techniques exploiting novel architectural features of emerging embedded processors. However, the lower code quality of the compilers is usually acceptable considering that the C compiler is available early in the processor architecture exploration loop. Thus, once the ASIP architecture exploration phase has converged and an initial working compiler is available, it must be manually refined to a highly optimizing compiler or the application’s hot spots must be manually replaced by assembly programs – both are time-consuming tasks. One way to reduce the design effort is to provide retargetable optimizations for those architectural features that characterize a processor class, e.g., hardware multi-threading for network processors (NPU) [110]. In this way, retargetability and high code quality for this particular class of processors is achieved. For instance, retargetable software pipelining support is less useful for scalar architectures; however, it is a necessity for the class of VLIW processors, and for this class it can be designed in a retargetable fashion. This book contributes retargetable optimization techniques for two common ASIP features to further improve the code quality of retargetable compilers.

A retargetable assembler, linker, simulator, and profiler complete the required software development infrastructure. Needless to say that keeping all tools manually consistent during architecture exploration is a tedious and error-prone task. Additionally, they must also be adapted to modifications performed in the other design phases. As a result, different automated design methodologies for efficient embedded processor design have evolved. Two contemporary approaches are presented in the next section.

2.3 Design Methodologies

One solution to increase the design efficiency is to significantly restrict the design space of the processor. More specifically, such design environments are limited to a
2.3 Design Methodologies

Predefined processor template whose software tools and architecture can be configured to a certain extent (Fig. 2.2).

Prominent examples for this approach are the Xtensa [215] and the ARCl tangent [43] processor families. Considering that all configuration options are preverified and the number of possible processor configurations is limited, the final processor can be completely verified. However, this comes at the expense of a significantly reduced design space, which imposes certain limitations. The coarse partitioning of the design space makes it inherently difficult to conceive irregular architectures suited for several application domains. Furthermore, certain settings of the template may also turn out to be redundant or suboptimal, like memory interface or the register file architecture for instance. Another limitation is imposed by the support for custom instructions. Such instructions must be typically given in an HDL description, and hence cannot be directly utilized by the compiler.

Another, more flexible concept for ASIP design is based on architecture description languages (ADLs). Such languages have been established recently as a viable solution for an efficient ASIP design (Fig. 2.3). ADLs describe the processor on a higher abstraction level, e.g., instruction accurate or cycle accurate, to hide implementation details. One of the main contribution of such languages is the automatic generation of the software toolkit from a single ADL model of the processor. Advanced ADLs are even capable of generating the system interfaces and a synthesizable HDL model from the same specification. This eliminates the consistency problem of the traditional ASIP design flow since changes to the processor model directly lead to a new and consistent set of software tools and hardware implementation. In this way, they provide a systematic mechanism for a top-down design and validation of complex systems. The high degree of automation reduces the design
Early ADLs, such as ISPS [157], were used for the simulation, evaluation, and synthesis of computers and other digital systems. Contemporary ADLs can be classified into three categories [112] based on the kind of information an ADL can capture:

**Instruction-set centric:** Instruction-set-centric languages have been designed with the generation of an HLL compiler in mind. Consequently, such languages must capture the instruction-set behavior (i.e., syntax, coding, semantic) of the processor architecture, whereas the information about the detailed micro-architecture (i.e., pipeline stages, memories, buses, etc.) does not need to be included. However, it is hardly possible to generate HDL models from such specifications. Typical representatives for this kind of ADLs are nML [10, 141], ISDL [97], and CSDL [186].

**Architecture centric:** These kinds of ADLs capture the structure in terms of architectural components. Therefore, they are well-suited for processor synthesis. But on the other hand, these languages typically have a low abstraction level leading to a quite detailed architecture specification. Unfortunately, it is quite difficult, if not impossible, to extract compiler-relevant information (e.g., instruction’s semantic) from such informal models. Prominent examples for this category of ADLs are MIMOLA [235], UDL/I [264], and AIDL [254].

**Combination of both:** These so-called mixed-level description languages [13] describe both, the instruction-set behavior and the structure of the design. This enables the generation of software tools as well as a synthesizable hard-
ware model. However, capturing both information can lead to a huge description, which is difficult to maintain. Additionally, such languages can suffer from inconsistencies due to duplicated informations. Certain architectural aspects need to be described twice, e.g., once for compiler generation and once for processor synthesis. ADLs belonging to this group are MDes [134], RADL [155], FlexWare [207], MADL/OSM [275], EXPRESSION [201], and LISA [15].

Obviously, designing an ADL that captures all aspects of ASIP design in an unambiguous and consistent way is a challenging task. This is further aggravated by the fact that most ADLs have originally been designed to automate the generation of a particular component and have then been extended to address the other aspects. As a result, ADLs are often well-suited for the purpose they have been designed for, but impose major restrictions on, or are even incapable of the generation of the other components. This is true in particular for the generation of compiler and simulator. Therefore, a further focus of this book are methodologies to generate compiler and simulator from a single ADL specification without limiting its flexibility or architectural scope. A detailed discussion of different ADLs is given in Chapter 4.

2.4 Synopsis

• Finding the optimal balance between flexibility and performance requires the evaluation of different architectural alternatives.
• HLL compilers are needed in the exploration loop to cope with the growing amount of software and to avoid hardware/software mismatches.
• The widely employed retargetable compilers suffer from their lower code quality as compared to handwritten compilers or assembly code.
• For quick design space exploration methodologies using predefined processor templates or ADL descriptions are proposed.
• ADL support for the automatic generation of the complete software tool chain (in particular, compiler and simulator) is currently not satisfactory.
• The primary focus of this book is the generation of C compilers from ADL processor models and retargetable optimization techniques to narrow the code quality gap.