ADL-driven Methodologies for Design Automation of Embedded Processors

Prabhat Mishra and Aviral Shrivastava

This chapter describes various ADL-driven methodologies for the development of efficient and reliable embedded processors. The ADLs designed for a specific domain (such as DSP or VLIW) or for a specific purpose (such as simulation or compilation) can be compact, and it is possible to automatically generate efficient tools and hardwares. However, it is difficult to design an ADL for a wide variety of architectures to perform different tasks using the same specification. Generic ADLs require the support of powerful methodologies to generate high-quality results compared to domain-specific/task-specific ADLs. This chapter presents a comprehensive overview of all the supported methodologies. It describes the fundamental challenges and required techniques to support powerful methodologies in the presence of generic ADLs. Chapters 3-14 present specific methodologies supported by the respective ADLs.

This chapter is organized as follows. Section 2.1 outlines ADL-driven design space exploration using generated software tools and hardware prototypes. Section 2.2 presents retargetable compiler generation approaches. Section 2.3 describes ADL-driven simulator generation techniques. ADL-driven implementation (hardware models) generation is discussed in Section 2.4. Section 2.5 describes various techniques for verifying the specification as well as the implementation. Finally, Section 2.6 concludes this chapter.

2.1 DESIGN SPACE EXPLORATION

Embedded systems present a tremendous opportunity to customize designs by exploiting the application behavior. Rapid exploration and evaluation of candidate architectures are necessary due to time-to-market pressure and short product lifetimes. The ADLs are used to specify processor and memory architectures and generate software toolkit including compiler, simulator, assembler, profiler, and debugger. Fig. 2.1 shows a traditional ADL-based design space exploration flow.
The application programs are compiled and simulated, and the feedback is used to modify the ADL specification with the goal of finding the best possible architecture for a given set of application programs under various design constraints such as area, power, and performance. An ADL-driven software toolkit generation enables performance-driven exploration. The simulator produces profiling data and thus may answer questions concerning the instruction set, the performance of an algorithm, and the required size of memory and registers. However, the required silicon area, clock frequency, and power consumption can only be determined by generating a synthesizable HDL (hardware) model.

One of the main purposes of an ADL is to support automatic generation of a high-quality software toolkit, including at least an ILP (instruction-level parallelism) compiler and a cycle-accurate simulator. However, such tools require detailed information about the processor, typically in a form that is not concise and easily specifiable. Therefore, it becomes necessary to develop procedures to automatically generate such tool-specific information from the ADL specification. For example, reservation tables (RTs) are used in many ILP compilers to describe resource conflicts. However, manual description of RTs on a per-instruction basis is cumbersome and error-prone. Instead, it is easier to specify the pipeline and datapath resources in an abstract manner, and generate RTs on a per-instruction basis [1]. Sections 2.2 and 2.3 describe some of the challenges in automatic generation of software tools (focusing on compilers and simulators) and survey some of the approaches adopted by current tools. Section 2.4 describes ADL-driven hardware generation approaches.

2.2 RETARGETABLE COMPILER GENERATION
The advent of System-on-Chip (SOC) technology resulted in a paradigm shift for the design process of embedded systems employing programmable processors with
custom hardware. Traditionally, embedded systems developers performed limited exploration of the design space using standard processor and memory architectures. Furthermore, software development was usually done using existing processors (with supported integrated software development environments) or done manually using processor-specific low-level languages (assembly). This was feasible because the software content in such systems was low, and also because the processor architecture was fairly simple (e.g., no instruction-level parallelism features) and well defined (e.g., no parameterizable components).

In the SOC domain, system-level design libraries increasingly consist of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIWs and hybrid ASIPs. These processor cores typically allow customization through parameterization of features (such as number of functional units, operation latencies, etc.). Furthermore, SOC technologies permit the incorporation of novel on-chip memory organizations (including the use of on-chip DRAM, frame buffers, streaming buffers, and partitioned register files). Together, these features allow exploration of a wide range of processor-memory organizations in order to customize the design for a specific embedded application.

An important and noticeable trend in the embedded SOC domain is the increasing migration of system functionality from hardware to software, resulting in a high degree of software content for newer SOC designs. This trend, combined with shrinking time-to-market cycles, has resulted in intense pressure to migrate the software development to a high-level language-based environment (such as C, C++, Java) in order to reduce the time spent in system design. To effectively explore the processor-memory design space and develop software in a high-level language, the designer requires a high-quality software toolkit (primarily a highly optimizing compiler and cycle-accurate simulator). Compilers for embedded systems have been the focus of several research efforts.

The compilation process can be broadly broken into two steps: analysis and synthesis [2]. During analysis, the program (in high-level language) is converted into an intermediate representation (IR) that contains all the desired information such as control and data dependences. During synthesis, the IR is transformed and optimized in order to generate efficient target-specific code. The synthesis step is more complex and typically includes the following phases: instruction selection, scheduling, resource allocation, code optimizations/transformations, and code generation. The effectiveness of each phase depends on the target architecture and the application. Additionally, a further problem during the synthesis step is that the optimal ordering between these phases (and other optimizations) is highly dependent on the target architecture and the application program. For example, the ordering between the memory assignment optimizations and instruction scheduling passes is very critical for memory-intensive applications. As a result, traditionally, compilers have been painstakingly hand-tuned to a particular architecture (or architecture class) and application domain(s). However, stringent time-to-market constraints for SOC designs no longer make it feasible to manually generate compilers tuned to particular architectures.
A promising approach to automatic compiler generation is the "retargetable compiler" approach. A compiler is classified as retargetable if it can be adapted to generate code for different target processors with significant reuse of the compiler source code. Retargetability is typically achieved by providing target machine information (in an ADL) as input to the compiler along with the program corresponding to the application. The origins of retargetable compiler technology can be traced back to UNCOL (Universal Computer-Oriented Language) project [3] where the key idea of separate front-ends and back-ends was suggested to reuse the middle-end of the compiler. According to this, all source code is converted to a common intermediate representation (CIR), on which the majority of code transformations/optimizations are performed. Once optimized, machine-specific back-end is used to generate the executable. This basic style is still extensively used in popular compilers like SUIF (Stanford University Intermediate Format), and GCC (GNU Compiler Collection). In the context of SOC design environments, the application specification remains the same, therefore, there is just one front-end; however, there will be processor-specific back-ends. Further research in this area has extended the concept of compiler code reuse. Several important compiler phases like instruction scheduling and register allocation must be performed in the back-end, and there is tremendous opportunity to reuse compiler code in them.

In the remainder of this section, we will classify ADL-based retargetable compilers in several ways to provide insight into the capabilities and differences of the various ADL-based retargetable compilers.

2.2.1 Retargetability Based on ADL Content
The first classification is based on the level of detail in which the processor architecture is described in the ADL.

Parameter-based retargetability
In this scheme, several microarchitectural parameters, including operation latencies, number of functional units, number of registers, and delay slots, are provided in the ADL. Retargetability in the compiler is provided by means of some compiler phases being parameterized on these microarchitectural parameters. This is the simplest form of retargetability, popularly used to retarget the register allocator by the number of general purpose registers and instruction scheduler by using operation latencies.

Structure-based retargetability
In this scheme, the structure of the processor is described as a netlist in the ADL. The compiler optimizes the code for the processor structure. This kind of retargetability is popularly used to describe the processor pipeline structure and perform detailed fine-grain instruction scheduling by modeling and avoiding resource and data hazards.
2.2 Retargetable Compiler Generation

Behavior-based retargetability

Retargeting the instruction selection requires the description of the semantics, or the behavior of instructions of the target processor in terms of instructions of the CIR (Common Intermediate Representation). In the general form of this approach, the mapping for instruction trees of CIR to instruction trees of the target processor is specified in the ADL, and the compiler effectively tiles the application in CIR using input tree patterns and replaces them with the equivalent target instruction tree patterns.

2.2.2 Retargetability Based on Compiler Phases

Another insightful way of classifying ADL-based retargetable compilers is according to the compiler phases which are retargetable. When compiling for a different processor, at least the instruction selection and the register allocation phases of the compiler must be modified; these must be retargetable in a “retargetable compiler”.

Retargetable instruction selection

The minimum information required to retarget instruction selection is the specification of the mapping of each CIR instruction in terms of the target instructions in the ADL. While this minimum information is sufficient, better instruction selection can be achieved if the ADL contains several mappings of CIR instructions to target instructions. The target instruction mappings can vary depending on the power, performance, code-size requirements, and the compiler can choose the mapping depending on its optimization metric. Several retargetable compilers use the IBurg [4] library to perform these complex mappings.

Retargetable register allocation

While simple retargetable register allocators are able to retarget if the number of general purpose registers is changed, in reality this is rarely the case. Register allocation is typically very closely tied to instruction selection. Very often, operands of instructions can only be mapped to a specific set of registers. The registers are therefore specified in groups of register classes, and the mapping of operands of an instruction to the register classes is specified in the ADL. Each variable in the application then gets associated with several register classes, depending on the appropriate instructions. Then the variable can be assigned to a register in the intersection of all the classes to which it belongs.

Retargetable instruction scheduling

At the basic level, the job of an instruction scheduler is to predict pipeline hazards, and find a legitimate reordering of instructions (that does not break data dependencies) to minimize them. The simplest form of retargetable instruction selection is when the instruction selection takes the operation latency of each operation, and generates a schedule using them. However, due to multiple pipelines, the delay of an operation may be dependent on the dependent operation itself.
Some retargetable compilers including GCC allow users to specify an automaton of instruction sequences that will minimize/avoid pipeline resource hazards. Instead of explicitly specifying the automata in the ADL, some retargetable compilers deduce reservation tables from the ADL, which describes the processor pipeline, and the flow of instructions through it. Reservation tables of instructions can be combined to detect all resource hazards in a schedule. In the most general case, operation tables can be extracted from the ADL if the pipeline structure of the processor and the flow of instructions and its operands are specified in the ADL.

### 2.2.3 Retargetability Based on Architectural Abstractions

The third and very insightful differentiating classification of various existing ADL-based retargetable compilers is based on the architectural abstractions that the compiler can provide retargetability for. Fig. 2.2 shows processor design abstractions for instruction set architecture, processor pipeline design, processor-memory interface, and the memory design. Existing retargetable compilers differ in the abstraction for which they provide retargetability.

**ISA retargetability**

Functional retargetability can be achieved by compiling for a new instruction set. Retargetability toward microarchitectural features is only required for optimizing the compiler. Instruction set retargetability requires retargetable instruction selection and register allocation. There are various examples of such retargetable compilers including AVIV [5] using ISDL, CHESS [6] using nML and Elcor [7]. One very interesting ISA retargetability is the ability of the EXPRESS [8] compiler to generate good code for “dual instruction set architectures”. Dual instruction set architectures typically have two instruction sets—one is the normal 32-bit instruction set, and the other is a narrow 16-bit wide instruction set. While the 32-bit ISA is a complete instruction set (IS), the narrow 16-bit ISA has only a compressed version of some of the most frequently used instructions. The idea is that if the whole application can be expressed only using the narrow instructions, then it would lead to a 50% reduction in the code size; however there are several challenges to achieve it. First,
not all instructions have mapping to the narrow IS, second, the narrow instructions can access only a fraction of the register file, and therefore indiscriminate conversion will lead to an increase in spilling, causing an increase in code size. Advanced compiler techniques of using a register pressure-based heuristic to determine the "regions of code to convert" provide consistently high degrees of code compression. The ADL-based retargetable compilers can perform this profitable conversion after describing the narrow instruction set and its mapping from the normal 32-bit instruction set in the ADL.

**Processor pipeline retargetability**

Processor pipeline retargetability implies that the compiler should be able to generate a good-quality code, even if the processor pipeline is modified. Compilers like GCC require the user to define instruction automatons, which include pipeline information, for example, instruction dependencies in them. While this information is directly usable by the compiler to perform instruction scheduling, it is not very convenient for the user. An SOC (System-on-Chip) developer has to understand the pipeline structure and the flow of operations in them, and then translate that information into instruction automatons. Very often, this is time consuming and an error-prone effort.

Advanced ADL-based retargetable compilers automatically derive the information needed by the compiler from a structural description of the processor pipeline. The processor pipeline is specified as a DAG of pipeline stages, and the flow of operations in the pipeline is indicated as an attribute of each pipeline stage. From this information, retargetable compilers can automatically generate Reservation Tables (RTs), even for multi-cycle and pipelined instructions. The RTs of instructions can be combined to detect all resource hazards, and therefore avoid resource hazards during instruction scheduling. There are various compilers in this category including ASIP Meister based on Cosy [9], LISA [10], MAML [11], MADL [12], GNR [13], and TIE [14].

Further developments in the ADL-based retargetable compiler technology allow the compilers to detect not only the resource hazards, but also data hazards, by automatically generating Operation Tables, or OTs [15], from the structural description of the processor pipeline and operation binding to the pipeline stages in the ADL description. Operation tables like RTs specify not only the resources that the operation uses in each cycle, but also what the instruction does with data in each cycle of its execution. Operation Tables can be used to accurately detect all resource and data hazards in a pipeline, even in the presence of partial bypassing in the processor.

**Memory hierarchy retargetability**

The significance of the memory hierarchy on both the runtime and the power consumption of the processor makes it extremely important for any retargetable compilers to know about the memory hierarchy, and optimize for it. Research on ADL-based, memory aware retargetable compilers can be divided into two parts—first is for cache-based systems, and second is for scratch-pad-based systems. For cache-based systems, initial research looked at code and data placement in the
memory to reduce conflict misses. The cache parameters like block size, associativity, and cache size are specified in the ADL, and the retargetable compiler places the data and code in the memory to reduce conflict misses.

The ADL-based retargetable compiler research has also looked at compiling for other cache hierarchies, for example, horizontally partitioned caches (HPCs). There are multiple (typically two) caches at the same level of memory hierarchy in this architecture, and data has to be exclusively divided among the two caches. By partitioning high and low temporal locality data, interference can be reduced. Recent research has shown that HPCs are very effective in reducing the energy consumption of the memory subsystem when one of the cache is small. The hierarchy of the caches is specified as a DAG (Directed Acyclic Graph), and the design parameters of each cache are specified. ADL-based retargetable compilers use this information to partition and map the data in the two caches. There are various compilers in this category including MIMOLA [16] and EXPRESS [17].

Significant research advances in memory-aware retargetable compilers has been made for scratch-pad-based systems. Scratch pads are (typically small) on-die memories, which unlike caches are mapped to a separate address space. Consequently, unlike caches, the use of scratch pad is explicit in the assembly. An assembly writer (a programmer, or a compiler) has to explicitly move data onto the scratch pad, use it from there, and then write it back to the memory, if needed. Techniques have been developed for using the scratch pad for global static data, heap data, and the local stack data. While the main benefit of using scratch pad is in the reduced power consumption due to the absence of miss-management hardware, the access times are much faster for scratch pads. The ADL-based retargetable compilers take the scratch pad size as an input, and are able to generate code to manage the scratch pad to gain power and performance improvements.

*Processor-memory interface retargetability*

With increasing memory latency, cache misses are becoming ever more important, and they pose interesting optimization challenges. The question is: what can we do while the processor is stalled, waiting for data from memory. For example, how do we minimize the energy consumption of the processor while it is stalled. If we wanted to switch to a low-power mode, it is not possible, because even to switch to the closest low-power mode, for example, in the Intel XScale processor, it takes more than 180 processor cycles while the memory latency or cache miss latency is only 30 cycles. Compiler techniques have been proposed to aggregate several stalls and create a large stall during which the processor can be profitably switched to low-power mode. Processor memory interface parameters are specified in the ADL description, and the processor can perform code transformations to enable the aggregation. The EXPRESS [18] compiler using EXPRESSION ADL supports retargetability based on processor-memory interface details.

The ADL-based retargetable compilers are able to automatically construct data structures containing information they need for their phases from the structural and
behavioral description of the processor. Using that, they are now able to optimize for several architectural and microarchitectural features to generate good code; however, a lot remains to be done. The microarchitectural exploration space is limited only by human creativity, and many features open new doors for compiler optimization. As compared to a native compiler, a retargetable compiler fundamentally should have only a compilation-time penalty; however, in practice, because of the microarchitectural nuances and idiosyncrasies, a native compiler typically performs better than a retargetable compiler. However, this improvement of a native compiler comes more as an evolution, as compiler designers become more and more aware of microarchitectural complexities, and implement pointed heuristics to achieve code quality, as the maturing of a compiler takes a long time. Retargetable compilers are invaluable as the first cut compiler for a new processor, and are immensely useful in early design space exploration of processor architectures.

### 2.3 Retargetable Simulator Generation

Simulators are critical components of the exploration and software design toolkit for the system designer. They can be used to perform diverse tasks such as verifying the functionality and/or timing behavior of the system (including hardware and software), and generating quantitative measurements (e.g., power consumption), which can be used to aid the design process. Simulation of the processor system can be performed at various abstraction levels. At the highest level of abstraction, a functional simulation of the processor can be performed by modeling only the instruction set (IS). Such simulators are termed instruction-set simulators (ISS) or instruction-level simulators (ILS). At lower levels of abstraction are the cycle-accurate and phase-accurate simulation models that yield more detailed timing information. Simulators can be further classified based on whether they provide bit-accurate models, pin-accurate models, exact pipeline models, or structural models of the processor.

Typically, simulators at higher levels of abstraction (e.g., ISS, ILS) are faster, but gather less information as compared to those at lower levels of abstraction (e.g., cycle-accurate, phase-accurate). Retargetability (i.e., the ability to simulate a wide variety of target processors) is especially important in the arena of embedded SOC design with emphasis on the exploration and co-design of hardware and software. Simulators with limited retargetability are very fast but may not be useful in all aspects of the design process. Such simulators typically incorporate a fixed architecture template and allow only limited retargetability in the form of parameters such as number of registers and ALUs. Examples of such simulators are numerous in the industry and include the HPL-PD [7] simulator using the MDES ADL. The model of simulation adopted has a significant impact on the simulation speed and flexibility of the simulator. Based on the simulation model, simulators can be classified into three types: interpretive, compiled, and mixed.
2.3.1 Interpretive Simulation

Such simulators are based on an interpretive model of the processor's instruction set. Interpretive simulators store the state of the target processor in host memory. It then follows a fetch, decode, and execute model: instructions are fetched from memory, decoded, and then executed in serial order as shown in Fig. 2.3. Advantages of this model include ease of implementation, flexibility, and the ability to collect varied processor state information. However, it suffers from significant performance degradation as compared to the other approaches, primarily due to the tremendous overhead in fetching, decoding, and dispatching instructions. Almost all commercially available simulators are interpretive. Examples of research interpretive retargetable simulators include SIMPRESS [19] using EXPRESSION, and GENSIM/XSIM [20] using ISDL.

2.3.2 Compiled Simulation

Compilation-based approaches reduce the runtime overhead by translating each target instruction into a series of host machine instructions which manipulate the simulated machine state, as shown in Fig. 2.4. Such translation can be done either at compile time (static compiled simulation) where the fetch-decode-dispatch overhead is completely eliminated, or at load time (dynamic compiled simulation), which amortizes the overhead over repeated execution of code. Simulators based on the static compilation model are presented by Zhu et al. [21] and Pees et al. [22]. Examples of dynamic compiled code simulators include the Shade simulator [23], and the Embra simulator [24].

2.3.3 Mixed Approaches

Traditional interpretive simulation is flexible but slow. Instruction decoding is a time-consuming process in a software simulation. Compiled simulation performs compile time decoding of application programs to improve the simulation performance. However, all compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is further more runtime static. Due to the restrictiveness of the compiled technique, interpretive simulators
are typically used in embedded systems design flow. Two recently proposed simulation techniques (JIT-CCS [25] and IS-CS [26]) combine the flexibility of interpretive simulation with the speed of the compiled simulation.

The just-in-time cache compiled simulation (JIT-CCS) technique compiles an instruction during runtime, just-in-time before the instruction is going to be executed. Subsequently, the extracted information is stored in a simulation cache for direct reuse in a repeated execution of the program address. The simulator recognizes if the program code of a previously executed address has changed, and initiates a recompilation. The instruction set compiled simulation (IS-CS) technique performs time-consuming instruction decoding during compile time. In case an instruction is modified at runtime, the instruction is re-decoded prior to execution. It also uses an instruction abstraction technique to generate aggressively optimized decoded instructions that further improve simulation performance [26, 27].

2.4 ARCHITECTURE SYNTHESIS

There are two major approaches in the literature for synthesizable HDL generation. The first one is a parameterized processor-core-based approach. These cores are bound to a single processor template whose architecture and tools can be modified to a certain degree. The second approach is based on processor specification languages.

2.4.1 Implementation Generation Using Processor Templates

Examples of processor-template-based approaches are Xtensa [14], Jazz [28], and PEAS-I [29, 30]. Xtensa [14] is a scalable RISC processor core. Configuration options include the width of the register set, caches, and memories. New functional units and instructions can be added using the Tensilica Instruction Language (TIE). A synthesizable hardware model along with software toolkit can be generated for this class of architectures. Improv’s Jazz [28] processor is supported
by a flexible design methodology to customize the computational resources and instruction set of the processor. It allows modifications of data width, number of registers, depth of hardware task queue, and addition of custom functionality in Verilog. PEAS-I [29, 30] is a GUI-based hardware/software codesign framework. It generates HDL code along with a software toolkit. It has support for several architecture types and a library of configurable resources.

2.4.2 ADL-driven Implementation Generation

Fig. 2.1 shows a typical framework of processor description language-driven HDL generation and exploration. The generated hardware models are also used for implementation validation, as described in Section 2.5. Structure-centric ADLs such as MIMOLA are suitable for hardware generation. Some of the behavioral languages (such as ISDL) are also used for hardware generation. For example, the HDL generator HGEN [20] uses ISDL description. Mixed languages such as nML, LISA, and EXPRESSION capture both the structure and behavior of the processor. The synthesizable HDL generation approach based on LISA language [31] produces an HDL model of the architecture. The designer has the choice to generate a VHDL, Verilog, or SystemC representation of the target architecture [31]. Similarly, the synthesis tool GO [32] uses nML description to generate synthesizable RTL models in VHDL or Verilog. The HDL generation methodology presented by Mishra et al. [33] combines the advantages of the processor-template-based environments and the language-based specifications using EXPRESSION ADL. The MAML language allows RTL generation based on highly parameterizable templates written in VHDL. Itoh et al. (PEAS-III [34, 35]) have proposed a micro-operation-description-based synthesizable HDL generation. It can handle processor models with a hardware interlock mechanism and multi-cycle operations [34].

2.5 TOP-DOWN VALIDATION

Validation of microprocessors is one of the most complex and important tasks in the current System-on-Chip (SOC) design methodology. Fig. 2.5 shows a traditional architecture validation flow. The architect prepares an informal specification of the microprocessor in the form of a natural language such as English. The logic designer implements the modules in the register-transfer level (RTL). The RTL design is validated using a combination of simulation-based techniques and formal methods. One of the most important problems in today's processor design validation is the lack of a golden reference model that can be used for verifying the design at different levels of abstraction. Thus, many existing validation techniques employ a bottom-up approach to pipeline verification, where the functionality of an existing pipelined processor is, in essence, reverse-engineered from its RT-level implementation.

Mishra et al. [36] have presented an ADL-driven validation technique that is complementary to these bottom-up approaches. It leverages the system architect's
2.5 Top-down Validation

knowledge about the behavior of the processor/memory architectures through ADL constructs, thereby allowing a powerful top-down approach to microprocessor validation. Fig. 2.6 shows an ADL-driven top-down validation methodology. This methodology has two important steps: validation of ADL specification, and ADL-driven validation of microprocessors.

2.5.1 Validation of ADL Specification

One of the most important requirements in a top-down validation methodology is to ensure that the specification (reference model) is golden. This section presents techniques to validate the static and dynamic behaviors of the architecture specified in an ADL. It is necessary to validate the ADL specification to ensure the correctness
CHAPTER 2 ADL-driven Methodologies

Architecture Specification
(Processor Manual/Document)

Automatic
Manual

SimGen

TestGen

HDLGen

Simulator
(Reference)

RTL Design
(Implementation)

ADL SPECIFICATION
(Golden Reference Model)

Verify Specification

Equivalence
9

Properties

Symbolic

Simulation

Equivalence Checking

RTL Design

FIGURE 2.6

Top-down validation flow.

of both the architecture specified and the generated executable models including a software toolkit and hardware implementation. The benefits of validation are two fold. First, the process of any specification is error-prone and thus verification techniques can be used to check for the correctness and consistency of the specification. Second, changes made to the processor during exploration may result in incorrect execution of the system and verification techniques can be used to ensure correctness of the modified architecture.

One of the major challenges in validating the ADL specification is to verify the pipeline behavior in the presence of hazards and multiple exceptions. There are many important properties that need to be verified to validate the pipeline behavior. For example, it is necessary to verify that each operation in the instruction set can execute correctly in the processor pipeline. It is also necessary to ensure that execution of each operation is completed in a finite amount of time. Similarly, it is important to verify the execution style of the architecture.

Typical validation scenario requires two models: specification (assumed to be correct) and implementation (needs to be verified). A set of tests can be applied on both specification (or its simulatable model) and implementation, and the corresponding outputs can be compared. However, in case of specification validation, we have only one model that needs to be verified. As a result, property checking (or model checking) is very suitable for specification validation. Property
Validation of ADL specification.

Checking ensures that a specification model satisfies a set of properties (or intended behaviors). Fig. 2.7 shows a property checking-based flow for specification validation. The ADL specification cannot be directly used for property checking. Therefore, it is necessary to generate executable formal models (such as graph or FSM models) from the specification. It is also required to generate a set of properties (behaviors) from the specification. The properties can be applied on the generated models using a model checking framework. The generated models and properties need to be suitable for the model checking framework. For example, a property can be a function that operates on a graph model [37]. In case a model checking-based framework such as SMV (symbolic model verifier) [38] is used, the model as well as the properties need to be specified in a specific SMV/temporal language.

Chapter 6 presents two property checking approaches for specification validation using EXPRESSION ADL. The first approach verifies a set of static behaviors/properties including connectedness, false pipeline and data-transfer paths, and completeness using a graph-based model of the architecture [37]. The second approach generates an FSM model from the specification to verify dynamic behaviors such as determinism and in-order execution in the presence of hazards and multiple exceptions [39]. The validated ADL specification can be used as a golden reference model for top-down validation of programmable architectures consisting of processor cores, coprocessors, and memory subsystems.

### 2.5.2 Implementation Validation

The ADL-driven validation approach has been demonstrated in two directions: simulation-based validation using directed test generation, and design validation using a combination of equivalence checking and symbolic simulation.

**Simulation-based validation using directed test generation**

Existing processor validation techniques employ a combination of simulation-based techniques and formal methods. Simulation is the most widely used form
of processor validation. Use of ADL specification improves the overall validation effort, since both simulator and directed tests can be automatically generated from the ADL specification, as shown in Fig. 2.6. The generated tests can be applied on the hardware implementation as well as on the generated cycle-accurate simulator (reference model), and the outputs can be compared to check the correctness of the implementation. Since ADL specification can be used to generate simulation models at different levels of abstraction, the same validation methodology can be used for verifying simulators. For example, an instruction-set simulator (reference model) can be used to verify a cycle-accurate simulator (implementation).

Various types of test programs are used during simulation: random, constrained-random, and directed tests. The directed test vectors are generated based on certain coverage metrics such as pipeline coverage, functional coverage, and so on. Directed tests are very promising in reducing the validation time and effort, since a significantly less number of directed tests are required compared to random tests to obtain the same coverage goal. Test generation for functional validation of processors has been demonstrated using MIMOLA [16], EXPRESSION [40], nML [32], and LISA [41]. The basic idea is to generate the required behaviors and constraints from the ADL specification to direct the test generation process. For example, the model checking-based approach is used in EXPRESSION framework [40, 42]. This approach generates a graph-based model of the pipelined processor. Based on the graph coverage, a set of properties/behaviors are generated from the specification. Finally, the negated version of the properties are applied on the design using a model checker. The model checker produces a set of counterexamples which are converted into test programs consisting of instruction sequences.

**Property/model checking**

Fig. 2.8 shows an ADL-driven property checking methodology. The basic idea is to generate the properties based on a certain coverage metric such as pipeline coverage, functional coverage, etc. These properties and RTL implementation cannot be directly applied to a model checker due to capacity restrictions. Instead, an abstracted version of the RTL implementation is used to reduce space complexity in model checking. However, the process of abstraction may introduce errors (false negative) or may suppress errors (false positive) present in the RTL implementation. Symbolic simulation can be used to apply properties directly on the RTL implementation [43]. Symbolic simulation combines traditional simulation with formal symbolic manipulation [44]. Each symbolic value represents a signal value for different operating conditions, parameterized in terms of a set of symbolic Boolean variables. By this encoding, a single symbolic simulation run can cover many conditions that would require multiple runs of a traditional simulator.

**Equivalence checking**

Equivalence checking is a branch of static verification that employs formal techniques to prove that two versions of a design either are or are not functionally equivalent. Fig. 2.9 shows an ADL-driven approach that uses the generated hardware
2.5 Top-down Validation

FIGURE 2.8
ADL-driven property checking.

FIGURE 2.9
ADL-driven equivalence checking.
prototype as a reference model for equivalence checking with the RTL implement-
ation [43]. An equivalence checker will try to match the compare points between
the designs. The unmatched compare points need to be mapped manually. The tool
tries to establish equivalence for each matched compare point. In case of failure,
the failing compare points are analyzed to verify whether they are actual failures or
not. The feedback is used to perform additional setup (in case of a false negative),
or to modify the implementation (RTL design).

The ADL-driven hardware generation and validation of design implementation
using equivalence checking has one limitation: the structure of the generated hard-
ware model (reference) needs to be similar to that of the implementation. This
requirement is primarily due the limitation of the equivalence checkers available
today. Equivalence checking is not possible using these tools if the reference and
implementation designs are large and drastically different. Property checking can
be useful in such scenarios to ensure that both designs satisfy a set of properties.
However, property checking does not guarantee equivalence between two designs.
As a result, it is also necessary to use other complementary validation techniques
(such as simulation) to verify the implementation.

2.6 CONCLUSIONS

Design of embedded systems presents a tremendous opportunity to customize the
implementation by exploiting the application behavior. Architecture Description
Languages have been used successfully to capture a wide variety of architectures, and
automatically generate software toolkits including the compiler, simulator, assembler,
and debugger. The generated tools allow efficient exploration of design alternatives
to determine the best possible architecture under various constraints such as area,
power, and performance. The ADL specification is also used to generate hardware
models as well as functional test patterns to enable top-down validation of embed-
ded processors. The ADL-driven methodologies reduce time-to-market and enable
generation of cost-effective and reliable embedded systems.

REFERENCES

ration of reservation tables from architectural descriptions. IEEE Transactions on Very Large
1986.


