Architecture Description Language (ADL)-Driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs

PRABHAT MISHRA
University of Florida
and
AVIRAL SHRIVASTAVA and NIKIL DUTT
University of California, Irvine

Advances in semiconductor technology permit increasingly complex applications to be realized using programmable systems-on-chips (SOCs). Furthermore, shrinking time-to-market demands, coupled with the need for product versioning through software modification of SOC platforms, have led to a significant increase in the software content of these SOCs. However, designer productivity is greatly hampered by the lack of automated software generation tools for the exploration and evaluation of different architectural configurations. Traditional hardware-software codesign flows do not support effective exploration and customization of the embedded processors used in programmable SOCs. The inherently application-specific nature of embedded processors and the stringent area, power, and performance constraints in embedded systems design critically require a fast and automated architecture exploration methodology. Architecture description language (ADL)-Driven design space exploration and software toolkit generation strategies present a viable solution to this problem, providing a systematic mechanism for a top-down design and validation of complex systems. The heart of this approach lies in the ability to automatically generate a software toolkit that includes an architecture-sensitive compiler, a cycle-accurate simulator, assembler, debugger, and verification/validation tools. This article illustrates a software toolkit generation methodology using the EXPRESSION ADL. Our exploration studies demonstrate the need for and usefulness of this approach, using as an example the problem of compiler-in-the-loop design space exploration of reduced instruction-set embedded processor architectures.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors; I.6.7 [**Simulation and Modeling**]: Simulation Support Systems

General Terms: Design, Languages

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712.

Authors' addresses: P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611; email: prabhat@cise.ufl.edu; A. Shrivastava, N. Dutt, Center for Embedded Computer Systems, Donald Bren School of Information and Computer Sciences, University of California, Irvine, CA 92697; email: {aviral, dutt}@ics.uci.edu.

Additional Key Words and Phrases: Architecture description language, design space exploration, programmable architecture, retargetable compilation, embedded processor

1. INTRODUCTION

Increasing complex system functionality and advances in semiconductor technology are changing how electronic systems are designed and implemented today. Escalating nonrecurring engineering (NRE) costs to design and manufacture chips have tilted the balance towards achieving greater design reuse. As a result, hardwired application-specific integrated circuit (ASIC) solutions are no longer attractive. Increasingly, we are seeing a shift toward systems implemented using programmable platforms. Furthermore, the high degree of integration provided by current semiconductor technology has enabled the realization of the entire system functionality onto a single chip, which we call a *programmable system-on-chip (SOC)*. Programmable SOCs are an attractive option not only because they provide a high degree of design reuse via software, but because they also greatly reduce the time-to-market.

With both system complexity and time-to-market becoming the main hurdles for design success, a key factor in programmable SOC design is the designer's productivity, that is, a designer's ability to quickly and efficiently map applications to SOC implementations. Furthermore, the need for product differentiation necessitates careful customization of the programmable SOC—a task that customarily takes a long time. 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 off-the-shelf processors (with supported integrated software development environments) or done manually using processor-specific low-level languages (e.g., assembly). This was feasible because the software content in such systems was low and the embedded processor architectures were fairly simple (e.g., no instruction-level parallelism) and well-defined (e.g., no parameterizable components). The emergence of complex, programmable SOCs poses new challenges for design space exploration. To enable efficient and effective design space exploration, the system designer critically needs methodologies that permit: (i) rapid tuning of the embedded processors for target applications, and (ii) automatic generation of customized software for the tuned embedded processors.

Figure 1 describes a contemporary hardware-software codesign methodology for the design of traditional embedded systems consisting of programmable processors, application-specific integrated circuits (ASICs), memories, and I/O interfaces [Mishra and Dutt 2004a]. This contemporary design flow starts by specifying an application in a system design language. The application is then partitioned into tasks that are either assigned to software (i.e., executed on the processor) or hardware (ASIC) such that design constraints (e.g., performance, power consumption, cost, etc.) are satisfied. After hardware-software partitioning, tasks assigned to software are translated into programs (either in high-level languages such as C/C++ or in assembly), and then compiled into

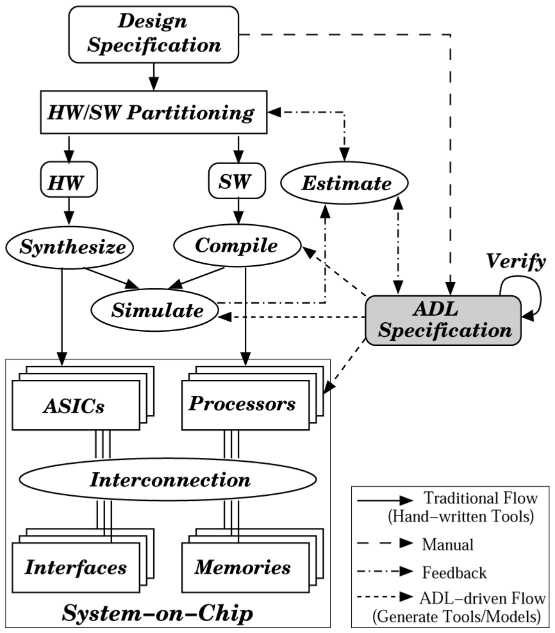


Fig. 1. Hardware-Software codesign flow for SOC design.

object code (which resides in memory). Tasks assigned to hardware are translated into hardware description language (HDL) descriptions and then synthesized into ASICs.

In traditional hardware-software codesign, the target architecture template is predefined. Specifically, the processor is fixed or can be selected from a library of predesigned processors, but customization of the processor architecture is not allowed. Even in codesign frameworks allowing customization of the processor, the fundamental architecture can rarely be changed. However, the inherently application-specific nature of the embedded processor and strict multidimensional design constraints (power, performance, cost, weight, etc.) critically require customization and optimization of the design, including processor design, memory design, and processor-memory organizations. The contemporary codesign flow (which does not permit much customization) limits the ability of the system designer to fully utilize emerging IP libraries, and furthermore, restricts the exploration of alternative (often superior) SOC architectures. Consequently, there is tremendous interest in a language-based design methodology for embedded SOC optimization and exploration.

Architectural description languages (ADLs) are used to drive design space exploration and automatic compiler/simulator toolkit generation. As with an HDL-Based ASIC design flow, several benefits accrue from a language-based design methodology for embedded SOC design, including the abilities to perform (formal) top-down verification and consistency checking, to easily modify the target architecture and memory organization for design space exploration, and to automatically generate the software toolkit from a single specification.

Figure 1 illustrates the ADL-Based SOC codesign flow, wherein the architecture template of the SOC (possibly using IP blocks) is specified in an ADL. This template is then validated to ensure that the specification is golden. The validated ADL specification is used to automatically generate a software toolkit to enable software compilation and cosimulation of the hardware and software. Another 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 an intense pressure to migrate the software development to a high-level language (such as C, C++, Java)-based environment in order to reduce the time spent in system design. To effectively perform these tasks, the SOC designer requires a high-quality software toolkit that allows exploration of a variety of processor cores (including RISC, DSP, VLIW, Superscalar, and ASIP) along with generation of optimized software, to meet stringent performance, power, code density, and cost constraints. Manual development of the toolkit is too time-consuming to be a viable option. An effective embedded SOC codesign flow must therefore support automatic software toolkit generation, without loss of optimizing efficiency. Such software toolkits typically include instruction-level parallelizing (ILP) compilers, cycle-accurate and/or instruction-set simulators, assemblers/dis-assemblers, profilers, debuggers, etc. In order to automatically generate these tools, software toolkit generators accept as input a description of the target processor-memory system specified in an ADL.

This article focuses on ADL-Driven software toolkit generation and design space exploration and uses the EXPRESSION project (http://www.ics.uci.edu/˜express) [Halambi et al. 1999] as an example to illustrate this methodology. Figure 2 shows a simplified methodology for ADL-Driven exploration. This methodology consists of four steps: design specification, validation of the specification, retargetable software toolkit generation, and design space exploration. The first step is to capture the programmable architecture using an ADL. The next step is to verify the specification to ensure the correctness of the specified architecture. The validated specification is used to generate a retargetable software toolkit that includes a compiler and a simulator. The generated software toolkit enables design space exploration of programmable architectures for given application programs under various design constraints such as area, power, and performance. A key feature of this ADL-Driven exploration framework is the use of a "compiler-in-the-loop" design space exploration (DSE) framework that allows designers to meaningfully and rapidly explore the design space by accurately tracking the impact of architectural modifications by the designer. In particular, the compiler in Figure 2 is an "exploration compiler" that can be used by architects to perform quantitative tradeoffs between various design metrics (e.g., power and performance) very early in the design process.

The rest of the article is organized as follows. Section 2 briefly surveys current ADLs and describes how to capture processor, coprocessor, and memory architectures using the EXPRESSION ADL. Section 3 presents validation techniques to verify the ADL specification. Section 4 presents the methodology for

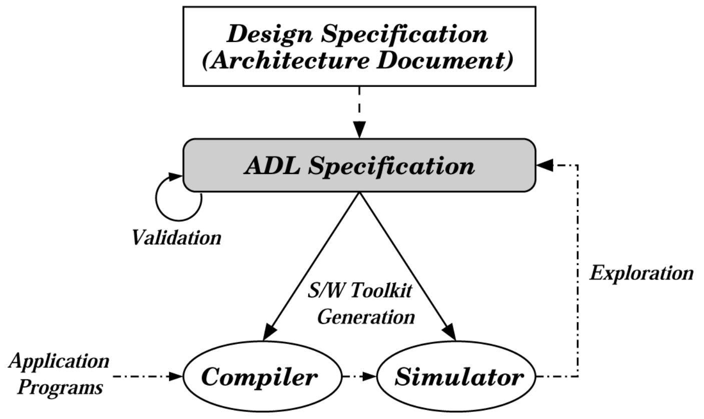


Fig. 2. ADL-Driven compiler-in-the-loop design space exploration.

retargetable compiler generation in the context of our compiler-in-the-loop exploration methodology. The retargetable simulator generation approach is described in Section 5, followed by a case study in Section 6. Finally, Section 7 concludes the article.

2. ARCHITECTURE-SPECIFICATION USING ADL

2.1 Brief Survey of ADLs

The phrase "architecture description language" (ADL) has been used in the context of designing both software and hardware architectures. Software ADLs are used for representing and analyzing software architectures [Clements 1996]. These ADLs capture the behavioral specifications of components and their interactions that comprise the software architecture. However, hardware ADLs capture the structure (hardware components and their connectivity) and behavior (instruction-set) of programmable architectures consisting of the processor, coprocessor, and the memory subsystem. Computer architects have long used machine description languages for the specification of architectures. Early ADLs such as ISPS [Barbacci 1981] were used for the simulation, evaluation, and synthesis of computers and other digital systems. Contemporary ADLs can be classified into three categories based on the nature of the information an ADL can capture: structural, behavioral, and mixed. Structural ADLs (e.g., MIMOLA [Leupers and Marwedel 1998]) capture the structure in terms of architectural components and their connectivity. Behavioral ADLs (e.g., nML [Freericks 1993] and ISDL [Hadjiyiannis et al. 1997]) capture the instruction-set behavior of the processor architecture. Mixed ADLs (e.g., LISA [Zivojnovic et al. 1996] and EXPRESSION [Halambi et al. 1999]) capture both the structure and behavior of the architecture. There are many comprehensive ADL surveys available in the literature, including ADLs for retargetable compilation [Qin and Malik 2002], SOC design [Tomiyama et al. 1999], and programmable embedded systems [Mishra and Dutt 2005]. Additionally, several commercial offerings following this ADL approach are now available, for example, Tensilica [Tensilica Inc. 2006] and LISATek (http://www.coware.com).

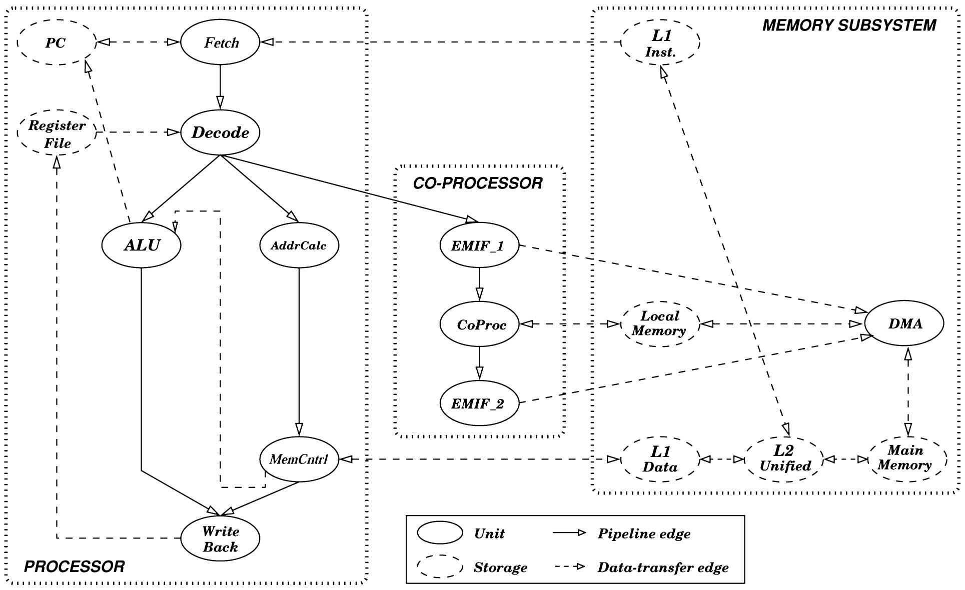


Fig. 3. An example architecture.

2.2 Specification Using EXPRESSION ADL

Our exploration framework uses the EXPRESSION ADL [Halambi et al. 1999] to specify processor, coprocessor, and memory architectures. Figure 3 shows an example of an architecture that can issue up to three operations (an ALU operation, a memory access operation, and a coprocessor operation) per cycle. The coprocessor supports vector arithmetic operations. In the figure, oval boxes denote units, dotted ovals are storages, bold edges are pipeline edges, and dotted edges are data-transfer edges. A data-transfer edge transfers data between units and storages. A pipeline edge transfers instruction (operations) between two units. A path from a root node (e.g., Fetch) to a leaf node (e.g, WriteBack) consisting of units and pipeline edges is called a *pipeline path*. For example, {*Fetch, Decode, ALU, WriteBack*} is a pipeline path. A path from a unit to the main memory or register file consisting of storages and data-transfer edges is called a *data-transfer path*. For example, {*MemCntrl, L1, L2, MainMemory*} is a data-transfer path. This section describes how the EXPRESSION ADL captures the structure and behavior of the architecture shown in Figure 3 [Mishra and Dutt 2004a].

2.2.1 *Structure.* The structure of an architecture can be viewed as a net-list with the components as nodes and the connectivity as edges. Similar to the processor's block diagram in its databook, Figure 4 shows a portion of the EXPRESSION description of the processor structure [Mishra and Dutt 2005]. It describes all the components in the structure, such as *PC*, *RegisterFile*, *Fetch*, *Decode*, *ALU*, and so on. Each component has a list of attributes. For example, the *ALU* unit has information regarding the number of instructions executed per cycle, the timing of each instruction, supported opcodes, input/output

```
# Components specification
( FetchUnit Fetch
  (capacity 3) (timing (all 1)) (opcodes all) ...) ...
)
( ExecUnit ALU
  (capacity 1) (timing (add 1) (sub 1) ...)
  (opcodes (add sub ...)) ...
)
( CPunit CoProc
  (capacity 1) (timing (vectAdd 4) ...)
  (opcodes (vectAdd vectMul))
)
......
# Pipeline and data-transfer paths
(pipeline Fetch Decode Execute WriteBack)
(Execute (parallel ALU LoadStore Coprocessor))
(LoadStore (pipeline AddrCalc MemCntrl))
(Coprocessor (pipeline EMIF_1 CoProc EMIF_2))
(dtpaths (WriteBack RegisterFile) (L1Data L2) ...)
......
# Storage section
( DCache L1Data
  (wordsize 64) (linesize 8) (associativity 2) ...
)
( ICache L1Inst (latency 1) ...)
( DCache L2 (latency 5) ...)
( DRAM MainMemory (latency 50) ...)
```

Fig. 4. Specification of structure using EXPRESSION ADL.

latches, and so on. Similarly, the memory subsystem structure is represented as a net-list of memory components. The memory components are described and attributed with their characteristics, such as cache line size, replacement policy, write policy, and so on.

The connectivity is established using the description of pipeline and data-transfer paths. For example, Figure 4 describes the four-stage pipeline as {*Fetch, Decode, Execute, WriteBack*}. The *Execute* stage is further described as three parallel execution paths: *ALU*, *LoadStore*, and *Coprocessor*. Furthermore, the *LoadStore* path is described using the pipeline stages *AddrCalc* and *MemCntrl*. Similarly, the coprocessor pipeline has three pipeline stages: *EMIF_1*, *CoProc*, and *EMIF_2*. The architecture has fifteen data-transfer paths, seven of which are unidirectional. For example, the path {*WriteBack* → *RegisterFile*} transfers data in one direction, whereas the path {*MemCntrl* ↔ *L1Data*} transfers data in both directions.

2.2.2 *Behavior.* EXPRESSION ADL captures the behavior of an architecture as the description of the instruction set and provides a programmer's view of the architecture. The behavior is organized into operation groups, with each group containing a set of operations[1] having some common characteristics. For example, Figure 5 [Mishra and Dutt 2005] shows three operation groups. The

[1]In this article we use the terms *operation* and *instruction* interchangeably.

```
# Behavior: description of instruction set
( opgroup aluOps (add, sub, . . . ) )
( opgroup memOps (load, store, . . . ) )
( opgroup cpOps (vectAdd, vectMul, . . . ) )
( opcode add
  (operands (s1 reg) (s2 reg/int16) (dst reg))
  (behavior dst = s1 + s2)
  (format 000101 dst(25-21) s1(21-16) s2(15-0))
)
( opcode store
  (operands (s1 reg) (s2 int16) (s3 reg))
  (behavior M[s1 + s2] = s3)
  (format 001101 s3(25-21) s1(21-16) s2(15-0))
)
( opcode vectMul
  (operands (s1 mem) (s2 mem) (dst mem) (length imm) )
  (behavior dst = s1 * s2)
)
. . .
# Operation Mapping
( target
  ( (addsub dest src1 src2 src3) )
)
( generic
  ( (add temp src1 src2) (sub dest src3 temp) )
)
. . .
```

Fig. 5. Specification of behavior using EXPRESSION ADL.

aluOps group includes all the operations supported by the *ALU* unit. Similarly, the *memOps* and *cpOps* groups contain all the operations supported by the units *MemCntrl* and *CoProc*, respectively. Each instruction is then described in terms of its opcode, operands, behavior, and instruction format. Each operand is classified either as source or as destination. Furthermore, each operand is associated with a type that describes the type and size of the data it contains. The instruction format describes the fields of the instruction in both binary and assembly. Figure 5 shows the description of *add*, *store*, and *vectMul* operations. Unlike normal instructions whose source and destination operands are register type (except load/store), the source and destination operands of *vectMul* are memory type. The *s1* and *s2* fields refer to the starting addresses of two source operands for the multiplication. Similarly, *dst* refers to the starting address of the destination operand. The *length* field refers to the vector length of the operation that has an immediate data type.

The ADL captures the mapping between the structure and the behavior (and vice versa). For example, the *add* and *sub* instructions are mapped to the *ALU* unit, the *load* and *store* instructions are mapped to the *MemCntrl* unit, and so on. The ADL also captures other information, such as mapping between the target and generic instruction set to enable retargetable compiler/simulator generation. For example, the target instruction *addsub* in Figure 5 is composed of the generic instructions *add* and *sub*. A detailed description of the EXPRESSION ADL is available in Halambi et al. [1999].

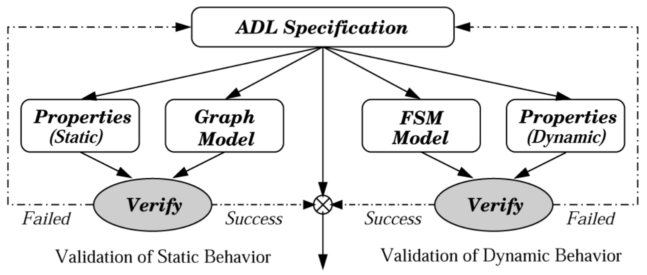


Fig. 6. Validation of ADL specification.

3. VALIDATION OF ADL SPECIFICATION

After specification of the entire programmable SOC architecture in an ADL, the next step is to verify the specification to ensure the correctness of the specified architecture. Although many challenges exist in specification validation, a particular challenge in pipelined architectures is to verify pipeline behavior in the presence of hazards and multiple exceptions. There are many important properties that need to be verified to validate 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 the execution of each operation is completed in a finite amount of time. Similarly, it is important to verify the execution style (e.g., in-order execution) of the architecture.

We have developed validation techniques to ensure that the architectural specification is well formed by analyzing both the static and dynamic behaviors of the specified architecture. Figure 6 shows the flow for verifying the ADL specification [Mishra and Dutt 2005]. The graph model as well as the FSM model of the architecture is generated from the specification. We have developed algorithms to verify several architectural properties, such as connectedness, false pipeline and data-transfer paths, completeness, and finiteness [Mishra and Dutt 2004a]. Dynamic behavior is verified by analyzing the instruction flow in the pipeline using a finite-state machine (FSM)-based model to validate several important architectural properties, such as determinism and in-order execution in the presence of hazards and multiple exceptions [Mishra et al. 2002, 2003]. The property checking framework determines if all the necessary properties are satisfied. In case of a failure, it generates traces so that a designer can modify the architecture specification.

4. RETARGETABLE COMPILER GENERATION

We now describe our approach for the generation of an architecture-exploration compiler from the EXPRESSION ADL. This approach is critical for the exploration of embedded systems that are characterized by stringent multidimensional constraints. To meet all the design constraints together, embedded systems very often have nonregular architectures. Traditional architectural features employed in high-performance computer systems need to be customized to meet the power, performance, cost, and weight needs of the embedded system. For example, an embedded system may not be able to implement complete register bypassing because of its impact on the area, power, and complexity of the system. As a result, the embedded system designer may opt for partial bypassing which can be customized to meet the system constraints. Further customization is possible in embedded systems due to their application-specific nature. Some architectural features which are not "very" useful for a given application set may be absent in the embedded processor.

In this highly customized world of embedded architectures, the role of the compiler is very crucial. The lack of regularity in the architecture poses significant challenges for compilers attempting to exploit these features. However, if the compiler is able to exploit these architectural features, it can have a significant impact on the power, performance, and other constraints of the whole system. As a result, compiler development is a very important phase of embedded processor design. A lot of time and effort on the part of experienced compiler-writers is invested in developing an optimizing compiler for the embedded processor. Given the significance the compiler has on processor power and performance, it is only logical that the compiler must play an important role in embedded processor design.

Although a compiler's effects can be incorporated into the design of an embedded processor, this process is often *ad hoc* in nature and may result in conservative, or worse yet, erroneous exploration results. For example, designers often use the code generated by the "old compiler," or "hand-generated" code to test new processor designs. This code should faithfully represent the code that the future compiler will generate. However, this approximation, or "hand-tuning," generates many inaccuracies in design exploration, and as a result may lead to suboptimal design decisions. A systematic method of incorporating compiler hints while designing the embedded processor is needed. Such a schema is called a compiler-assisted or compiler-in-the-loop design methodology. The key enabler of the compiler-in-the-loop methodology is an ADL-Driven retargetable compiler. While a conventional compiler takes only the application as input and generates the executable, a retargetable compiler also takes the processor architecture description as an input. The retargetable compiler can therefore exploit architectural features present in the described system, and generate code tuned to the specific architecture.

Whereas there has been a wealth of research on retargetable compilers [Muchnick 1997; MDES User Manual 1997], contemporary research on ADL-Driven retargetable compilers has focused on both design abstraction levels: architecture (instruction-set) and microarchitecture (implementation such as pipeline structure). Traditionally, there has been more extensive research on architecture-level retargetability. Our EXPRESSION-Based compiler-in-the-loop exploration framework employs and advances compiler retargetability at both abstraction levels. At the processor pipeline (microarchitecture) level, decisions on which bypasses should be present in the processor greatly impact the power, performance, and complexity of the processor. Indeed, our recent research results [Shrivastava et al. 2004, 2005; Park et al. 2006] show that deciding the bypasses in a processor by a traditional "simulation-only" exploration leads to incorrect design decisions and may lead to suboptimal designs. A

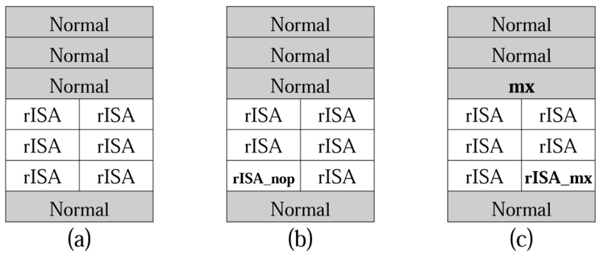


Fig. 7. rISA and normal instructions coreside in memory.

compiler-in-the-loop exploration can be used to suggest pareto-optimal bypass configurations.

In this section we will focus on the use of our compiler-in-the-loop exploration methodology at the architectural level, and investigate instruction-set architecture extensions for code size reduction.

4.1 Instruction-Set-Level Retargetability (rISA)

Many embedded systems are constrained by small memory footprints and thus require minimization of the program code size. An architectural feature designed to reduce code size is the reduced bit-width instruction set architecture (rISA) which supports two instruction sets: the normal set, which contains the original 32-bit instructions, and the reduced bit-width instruction-set, which encodes the most commonly used instructions into 16-bit instructions. If an application is fully expressed in terms of reduced bit-width instructions, then a 50% code size reduction is achieved (as compared to when it is expressed in terms of normal instructions).

Several embedded processors support this feature. For instance, the ARM processor supports rISA with 32-bit normal instructions and narrow 16-bit instructions. While the normal 32-bit instructions comprise the *ARM* instruction-set, the 16-bit instruction forms the *Thumb* instruction-set. Other processors with a similar feature include the MIPS32/16 [Kissell 1997], TinyRISC [LSI LOGIC 2006], STMicro's ST100 (http://www.st.com), and the ARC Tangent (http://www.arcores.com) processor.

The code for an rISA processor contains both normal and rISA instructions, as shown in Figure 7, from Shrivastava et al. [2006]. The fetch mechanism of the processor is oblivious to the processor's mode of execution regardless of the processor executing an rISA or normal instruction, the instruction fetch of the processor remains unchanged. The processor dynamically converts the rISA instructions to normal instructions before or during the instruction decode stage. Figure 8 shows the dynamic translation of Thumb instructions to ARM instructions, as described in [Advanced RISC Machines Ltd. 2006].

Typically, each rISA instruction has an equivalent instruction in the normal instruction-set. This is done to ensure that the translation from an rISA instruction to a normal instruction (which has to be performed dynamically) is simple, and can be done without performance penalty. After conversion, the instruction executes as a normal instruction, and thus no other hardware change

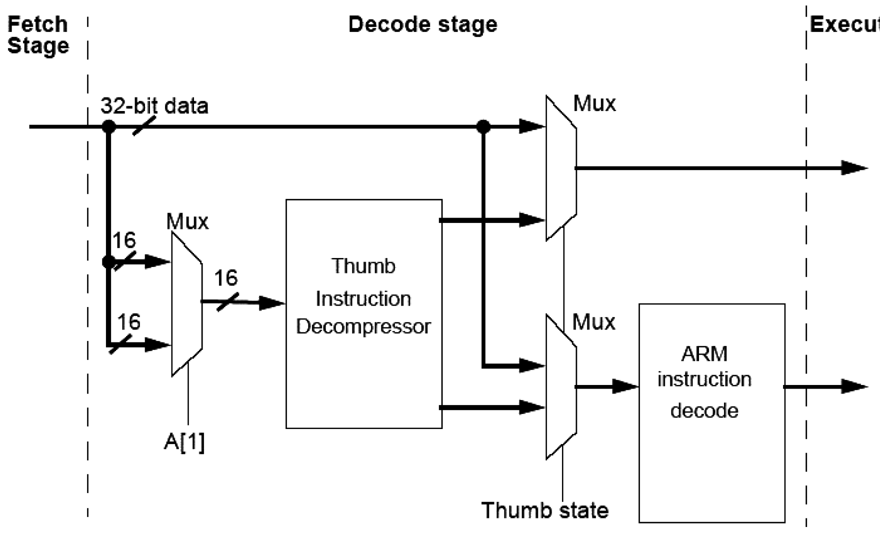


Fig. 8. Translation of Thumb instruction to ARM instruction.

is required. Therefore, the main advantage of rISA lies in achieving good code size reduction with minimal hardware additions. However, some instructions (e.g., an instruction with a long immediate operand) cannot be mapped into a single rISA instruction. It takes more than one rISA instruction to encode the normal instruction. In such cases, more rISA instructions are required to implement the same task. As a result, rISA code has slightly lower performance as compared to normal code.

The rISA instruction-set (IS), because of bit-width restrictions, can encode only a subset of the normal instructions and allows access to only a small subset of registers. Contemporary rISA processors (such as ARM/Thumb and MIPS32/16) incorporate a very simple rISA model with rISA instructions that can access 8 registers (out of 16 or 32 general-purpose registers). Owing to tight bit-width constraints in rISA instructions, they can use only very small immediate values. Furthermore, existing rISA compilers support the conversion only at a routine-level (or function-level) of granularity. Such severe restrictions make code size reduction obtainable by using a rISA very sensitive to the compiler quality and the application features. For example, if the application has high register pressure, or if the compiler does not do a good job of register allocation, it might be better to increase the number of accessible registers at the cost of encoding only a few opcodes in rISA. Thus, it is very important to perform a compiler-in-the-loop design space exploration (DSE) while designing rISA architectures.

4.1.1 *rISA Model.* The rISA model defines the rISA IS and the mapping of rISA instructions to normal instructions. Although typically each rISA instruction maps to only one normal instruction, there may be instances where multiple rISA instructions map to a single normal instruction.

The rISA processors can operate in two modes: rISA mode and normal mode. Most rISA processors (e.g., ARM) have a mode bit in the CPSR (current process

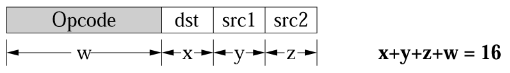


Fig. 9. Bit-width constraints on rISA instructions.

state register) which identifies whether the processor is in rISA mode or normal mode. When the processor is in rISA mode, it breaks the instruction into two halves and decodes them separately and sequentially. The processor needs to find out whether the instructions it is receiving are normal or rISA instructions. Many rISA processors accomplish this by using explicit instructions that change the mode of execution. We label an instruction in the normal IS that changes mode from normal to rISA the *mx* instruction, and an instruction in the rISA instruction-set that changes mode from rISA to normal the *rISA_mx* instruction. Every sequence of rISA instructions starts with an *mx* instruction and ends in a *rISA_mx* instruction. Such a sequence of rISA instructions is termed an *rISABlock*.

In order to avoid changing the instruction fetch mechanism of rISA processors, it is important to ensure that all normal instructions align to the word boundary. However, an odd number of instructions in a rISABlock results in the ensuing normal instruction being misaligned to the word boundary. Therefore, a padding *rISA_nop* instruction is required to force alignment to the word boundary.

Due to bit-width constraints, a rISA instruction can access only a subset of registers. The register accessibility of each register operand must be specified in the rISA model. The width of immediate fields must also be specified. In addition, there may be special instructions in the rISA model to help the compiler generate better code. For example, a very useful technique to increase the number of registers accessible in rISA mode is to implement a rISA move instruction that can access all registers.[2] Another technique to increase the size of the immediate operand value is to implement a rISA extend instruction that completes the immediate field of the succeeding instruction. Numerous such techniques can be explored to increase the efficacy of rISA architectures. Next, we describe some of the important rISA design parameters that can be explored using our framework.

4.1.2 *rISA Design Parameters.* The most important consequence of using rISA instructions is the limited number of bits available to encode the opcode, register operands, and the immediate values—resulting in a large space of alternative encodings that the rISA designer needs to explore and evaluate. For instance, in most existing architectures the register operand field in the rISA instructions is 3-bit wide, permitting access to only 8 registers. For 3-address instructions, $(3 \times 3) = 9$ bits are used in specifying the operands. Therefore, only $(16–9) = 7$ bits are left to specify the opcode. As a result, only $2^7 = 128$ opcodes can be encoded in rISA. The primary advantage of this approach is the huge number of opcodes available. Using such a rISA, most normal

[2]This is possible because a move has only two operands and hence, has more bits to address each operand.

32-bit instructions can be specified as rISA instructions. However, this approach suffers from the drawback of increased register pressure, possibly resulting in poor code size.

One modification is to increase the number of registers accessible by rISA instructions to 16. However, in this model only a limited number of opcodes are available. Thus, depending on the application, large sections of program code might not be implementable using rISA instructions. The design parameters that can be explored include the number of bits used to specify operands (and opcodes), and the type of opcodes that can be expressed in rISA.

Another important rISA feature that impacts the quality of the architecture is the "implicit operand format". In this feature, one (or more) of the operands in the rISA instruction is hard-coded (i.e., implied). The implied operand could be a register operand or a constant. When the frequently occurring case of a format of add instruction is $add\ R_i\ R_i\ R_j$ (where the first two operands are the same), an rISA instruction $rISA\_add1\ R_i\ R_j$ can be used. In the case of an application that access arrays produces a lot of instructions like $addr = addr + 4$, then an rISA instruction $rISA\_add4\ addr$, which has only one operand, might be very useful. The translation unit, while expanding the instruction can also fill in the missing operand fields. This is a very useful feature that can be used by the compiler to generate high-quality code.

Another severe limitation of rISA instructions is their inability to incorporate large immediate values. For example, with only 3 bits available for operands, the maximum unsigned value that can be expressed is 7. Thus, it might be useful to vary the size of the immediate field, depending on the application and the values that are (commonly) generated by the compiler. However, increasing the size of the immediate fields will reduce the number of bits available for opcodes (and also the other operands). This tradeoff can be meaningfully made only with a compiler-in-the-loop DSE framework.

Various other design parameters such as partitioned register files, shifted/padded immediate fields, etc. should also be explored in order to generate an rISA architecture that is tuned to the needs of the application and to the compiler quality. While some of these design parameters have been studied in a limited context, there is a critical need for an ADL-Driven framework that uses an architecture-aware compiler to exploit and combine all of these features. We now describe our rISA compiler framework which enables a compiler-in-the-loop exploration framework to quantify the impact of these features.

4.1.3 *rISA Compiler.* Figure 10 shows the phases of our EXPRESS compiler [Halambi et al. 2001] that are used to perform rISAization.[3] We use the GCC front end to output a sequence of generic 3-address MIPS instructions, which in turn are used to generate the CFG, DFG, and other compiler data structures comprising the internal representation of the application. We now describe each ensuing step of Figure 10.

Mark rISA Blocks. Due to the restrictions discussed in the previous subsection, several types of instructions such as those with very many operands,

[3]rISAization is the process of converting normal instructions to rISA instructions.

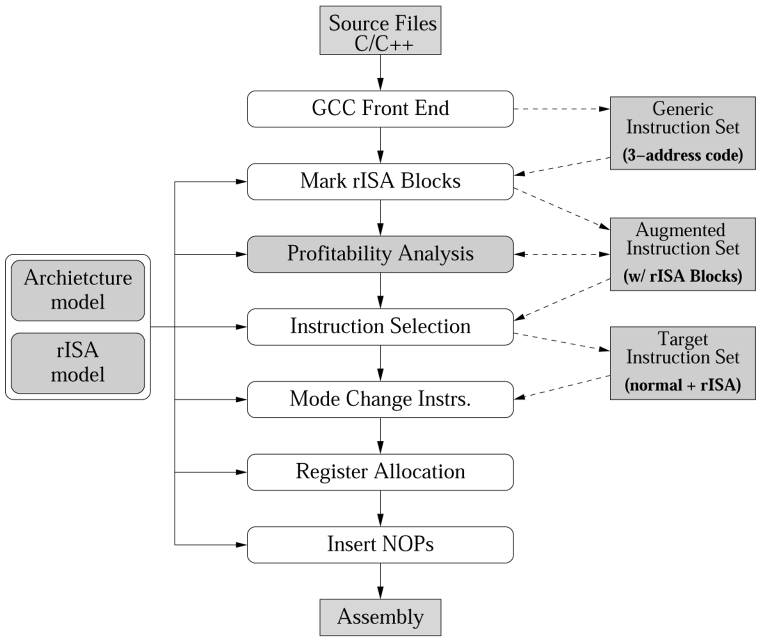


Fig. 10. rISA compiler flow for DSE.

or those with long immediate values, etc., may not be convertible to rISA instructions. The first step in compilation for a rISA processor is to mark all the instructions that can be converted into rISA instructions. A contiguous list of marked instructions in a basic block is termed a rISABlock. Owing to the overhead associated with rISAization, it may not be profitable to convert all rISABlocks into rISA instructions.

Profitability Analysis. This step decides which rISABlocks to convert into rISA instructions, and which ones to leave in terms of normal instructions. A mode change instruction is needed at the beginning and end of each rISA Block. Furthermore, in order to adhere to the word boundary, there should be an even number of instructions in each rISABlock. Therefore, if a rISABlock is very small, then the mode change instruction overhead could negate gains from code compression achieved through rISAization. It should be noted here that if all the predecessors of a basic block are also decided to be encoded in rISA mode, then the mode change instructions may not be needed. We will perform and discuss such optimizations at a later stage.

Similarly, if the rISABlock is very big, the increased register pressure (and the resulting register spills) could render rISAization unprofitable. Thus, an accurate estimation of code size and performance tradeoff is necessary before rISAizing a rISABlock. In our technique, the impact of rISAization on code size and performance is estimated using a profitability analysis function.

The profitability analysis function estimates the difference in code size and performance if the block were to be implemented in rISA mode as compared to normal mode. The compiler (or user) can then use these estimates to tradeoff between performance and code size benefits for the program. The profitability heuristic is described in greater detail in Halambi et al. [2002].

Instruction Selection. Our compiler uses a tree pattern matching-based algorithm for instruction selection. A tree of generic instructions is converted into a tree of target instructions. In case a tree of generic instructions is replaced by more than one target instruction tree, the one with lower cost is selected. The cost of a tree depends upon the user's perception of the relative importance of performance and code size.

Normally, during instruction selection a tree of generic instructions has trees of target instructions as possible mappings. Our rISA framework provides trees of rISA instructions also for possible mapping. As a result, the instruction selection for rISA instructions becomes a natural part of the normal instruction selection process. The instruction selection phase uses a profitability heuristic to guide decisions on which sections of a routine to convert to rISA instructions, and which to convert to normal target instructions. All generic instructions within profitable rISABlocks are replaced with rISA instructions, and all other instructions are replaced with normal target instructions.

Replacing a generic instruction with a rISA instruction involves two steps: first converting the generic instruction to the appropriate rISA opcode, and second, restricting the operand variables to the set of rISA registers. This is done by adding a restriction on the variable, which implies that this variable can be mapped to the set of rISA registers only.

Mode Change Instructions. After instruction selection, the program contains sequences of normal and rISA instructions. A list of contiguous rISA instructions may span across basic block boundaries. To ensure correct execution, we need to make sure that whenever there is a switch in the instructions from normal to rISA or vice versa (for any possible execution path), there is also an explicit and appropriate mode change instruction. There should be an *mx* instruction when the instructions change from normal to rISA instructions, and an *rISA_mx* instruction when the instructions change from rISA instructions to normal instructions. If the mode of instructions changes inside a basic block, then there is no choice but to add the appropriate mode change instruction at the boundary. However, when the mode changes at the basic block boundary, the mode change instruction can be added at the beginning of the successor basic block or at the end of the predecessor basic block. The problem becomes more complex if there is more than one successor and one predecessor at the junction. In such a case, the mode change instructions should be inserted so as to minimize performance degradation, that is, in the basic blocks which execute the least. We use a profile-based analysis to find where to insert the mode change instructions [Shrivastava and Dutt 2004].

Register Allocation. Our EXPRESS compiler implements a modified version of Chaitin's solution [Briggs et al. 1994] to register allocation. Registers are grouped into (possibly overlapping) register classes. Each program variable is then mapped to the appropriate register class. For example, operands of an rISA instruction belong to the rISA register class (which consists of only a subset of the available registers). The register allocator then builds the interference graph and colors it honoring the register class restrictions of the variables. Since code blocks that have been converted to rISA typically have a higher register pressure (due to the limited availability of registers), higher priority is given to rISA variables during register allocation.

Insert NOPs. The final step in code generation is to insert *rISA_nop* instruction in *rISABlocks* that have an odd number of rISA instructions. The output of the compiler flow in Figure 10 is the target assembly code exploiting the rISA instructions.

We have briefly described how the EXPRESSION ADL-Driven EXPRESS compiler can be used to generate architecture-sensitive code for rISA architectures. By considering the compiler effects during DSE, the designer is able to accurately estimate the impact of various rISA features. Section 6 presents some results for rISA exploration using the MIPS 32/16-bit rISA architecture.

5. RETARGETABLE SIMULATOR GENERATION

Simulators are indispensable tools in the development of new architectures. They are used to validate an architecture design, a compiler design as well as to evaluate architectural design decisions during design space exploration. Running on a host machine, these tools mimic the behavior of an application program on a target machine. These simulators should be fast to handle the increasing complexity of processors, flexible to handle all features of applications and processors, for example, runtime self-modifying codes, and retargetable to support a wide spectrum of architectures.

The performance of simulators varies widely depending on the amount of information they capture, as determined by their abstraction level. *Functional* simulators only capture the instruction-set details. The *cycle-accurate* simulators capture both instruction-set and microarchitecture details. As a result, cycle-accurate simulators are slower than functional simulators but provide timing details for the application program. Similarly, *bit-* and *phase-accurate* simulators model the architecture more accurately at the cost of simulation performance.

Simulators can be further classified based on their model of execution: either *interpretive* or *compiled*. Interpretive simulation is flexible but slow. In this technique an instruction is fetched, decoded, and executed at runtime. Instruction decoding is a time-consuming process in a software simulation. Compiled simulation performs compile time decoding of application program to improve the simulation performance. The performance improvement is obtained at the cost of flexibility. Compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is furthermore runtime static. Compiled simulators are not applicable to embedded systems that support runtime self-modifying codes or multimode processors. There are various ADL-Driven simulator generation frameworks in the literature including GENSIM using ISDL [Hadjiyiannis et al. 1999], MSSB/MSSU using MIMOLA [Leupers and Marwedel 1998], CHECKERS using nML [Freericks 1993], SIMPRESS/RexSim using EXPRESSION [Reshadi et al. 2003], HPL-PD using MDES [MDES User Manual 1997], and fast simulators

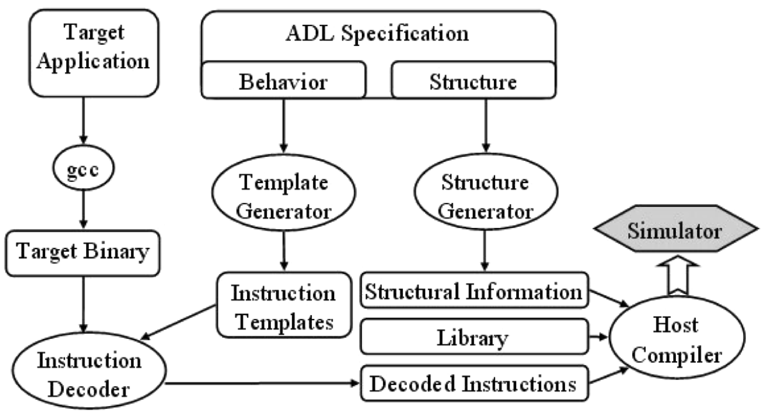


Fig. 11. ADL-driven instruction-set simulator generation.

using LISA [Pees et al. 2000] and RADL [Siska 1998]. This section briefly describes retargetable simulator generation using the EXPRESSION ADL for instruction-set architecture (ISA) simulation as well as cycle-accurate simulation.

5.1 Instruction-Set Simulation

In a retargetable ISA simulation framework, the range of architectures that can be captured and the performance of the generated simulators depend on three factors: first, the model, based on which the instructions are described; second, the decoding algorithm that uses the instruction model to decode the input binary program; and third, the execution method of decoded instructions. These issues are equally important, and ignoring any of them results in a simulator that is either very general but slow or very fast but restricted to some architecture domain. However, the instruction model significantly affects the complexity of decode and the quality of execution. We have developed a generic instruction model coupled with a simple decoding algorithm that leads to an efficient and flexible execution of decoded instructions [Reshadi et al. 2003, 2006].

Figure 11 shows our retargetable simulation framework that uses the ADL specification of the architecture and the application program binary (compiled by gcc) to generate the simulator [Reshadi et al. 2006]. Section 5.1.1 presents our generic instruction model used to describe the binary encoding and behavior of instructions [Reshadi et al. 2006]. Section 5.1.2 describes the instruction *decoder* that decodes the program binary using the description of instructions in the ADL. The decoder generates optimized source code of the decoded instructions [Reshadi et al. 2003]. The *structure generator* generates structural information to keep track of the state of the simulated processor. The target-independent components are described in the *library*. This library is finally combined with the structural *information* and the decoded instructions and is compiled on the host machine to get the final ISA simulator.

5.1.1 *Generic Instruction Model.* The focus of this model is on the complexities of different instruction binary formats in different architectures [Reshadi et al. 2006]. As an illustrative example, we model the integer arithmetic instructions of the Sparc V7 processor which is a single-issue processor with 32-bit instruction (http://www.sparc.org). The integer arithmetic instructions, *IntegerOps* (as shown below), perform certain arithmetic operations on two source operands and write the result to the destination operand. This subset of instructions is distinguished from the others by the following bit mask:

```
Bitmask:       10xxxxx0  xxxxxxxx  xxxxxxxx  xxxxxxxx
IntergerOps:   < opcode    dest      src1      src2 >
```

A bit mask is a string of '1,' '0,' and 'x' symbols and it matches the bit pattern of a binary instruction if and only if for each '1' or '0' in the mask, the binary instruction has a 1 or a 0 value in the corresponding position, respectively. The 'x' symbol matches both 1 and 0 values. In this model, an instruction of a processor is composed of a series of slots, $I =< sl_0, sl_1, \ldots >$, and each slot contains only one operation from a subset of operations. All the operations in an instruction execute in parallel. Each operation is distinguished by a mask pattern. Therefore, each slot (sl_i) contains a set of operation-mask pairs (op_i, m_i) and is defined by the format: $sl_i =< (op_i^0, mi_0)|(op_i^1, mi_1)| \ldots >$.

An operation class refers to a set of similar operations in the instruction-set that can appear in the same instruction slot and has a similar format. The previous slot description can be rewritten using an operation class clOps: $sl_i =< (clOps_i, m_i) >$. For example, integer arithmetic instructions in Sparc V7 can be grouped into a class (IntegerOps) as: $I_{SPARC} =< (IntegerOps, 10xx - xxx0xxxx - xxxxxxxx - xxxxxxxx - xxxx)| \ldots >$. An operation class is composed of a set of symbols and an expression that describes the behavior of the operation class in terms of the values of its symbols. For example, the operation class has four symbols: opcode, dest, src1, and src2. The expression for this example would be: dest = f_{opcode}(src1, src2). Each symbol may have a different type depending on the bit pattern of the operation instance in the program. For example, the possible types for the src2 symbol are register and immediate integer. The value of a symbol depends on its type and can be static or dynamic. For example, the value of a register symbol is dynamic and is known only at runtime, whereas the value of an immediate integer symbol is static and is known at compile time. Each symbol in an operation has a possible set of types. A general operation class is then defined as: $clOps =< (s_0, T_0), (s_1, T_1), \ldots |exp(s_0, s_1, \ldots) >$, where ($s_i$, T_i) are (symbol, type) pairs and exp(s_0, s_1,...) is the behavior of the operations based on the values of the symbols.

The type of a symbol can be defined as a register ($\in$ *Registers*) or as an immediate constant ($\in$ *Constants*), or can be based on certain microoperations ($\in$ *Operations*). For example, a data processing instruction in ARM (e.g., add) uses shift (microoperation) to compute the second source operand, known as ShifterOperand. Each possible type of a symbol is coupled with a mask

```
SPARCInst = $ (InegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) | .... $;
IntegerOp = <
    (opcode, OpTypes), (dest, DestType), (src1, Src1Type),  (src2, Src2Type)
    | {    dest = opcode(src1, src2); }
>;
OpTypes = {
    (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
    (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx),
    (Or , xxxx-xxxx 0010-xxxx xxxx-xxxx xxxx-xxxx),
    (And, xxxx-xxxx 0001-xxxx xxxx-xxxx xxxx-xxxx),
    (Xor, xxxx-xxxx 0011-xxxx xxxx-xxxx xxxx-xxxx),
    ....
};
DestType = [IntegerRegClass, 29, 25];
Src1Type = [IntegerRegClass, 18, 14];
Src2Type = {
    ([IntegerRegClass,4,0], xxxx-xxxx xxxx-xxxx xx0x-xxxx xxxx-xxxx),
    (\#int,12,0\#, xxxx-xxxx xxxx-xxxx xx1x-xxxx xxxx-xxxx)
};
```

Fig. 12. Description of integer arithmetic instructions in SPARC processor.

pattern that determines what bits in that operation must be checked to find out the actual type of the corresponding symbol. Possible types of a symbol are defined as: $T = \{(t, m) | t \in Operations \cup Registers \cup Constants, m \in (1|0|x)*\}$. For example, the opcode symbol can be any of the valid integer arithmetic operations (*OpTypes*), as shown in Figure 12.

Note that this provides more freedom for describing the operations because here the symbols are not directly mapped to some contiguous bits in the instruction, and a symbol can correspond to multiple bit positions in the instruction binary. The actual register in a processor is defined by its class and index. The index of a register in an instruction is defined by extracting a slice of the instruction bit pattern and interpreting it as an unsigned integer. An instruction can also use a specific register with a fixed index, as in a branch instruction that updates the program counter. A register is defined by: $r = [regClass, i, j] | [regClass, index]$, where i and j define the boundary of the index bit slice in the instruction. For example, if the dest symbol is from the 25th to 29th bits in the instruction and is an integer register, its type can be described as: $DestType = [IntegerRegClass, 29, 25]$. Similarly, a portion of an instruction may be considered as a constant. For example, one bit in an instruction can be equivalent to a Boolean type, or a set of bits can make an integer immediate. It is also possible to have constants with fixed values in the instructions. A constant type is defined by c = # type, ij # | # type, value # where i and j show the bit positions of the constant and the type is scalar, such as integer, Boolean, float, etc. The complete description of integer arithmetic instructions in the SPARC processor is shown in Figure 12.

5.1.2 *Generic Instruction Decoder.* A key requirement in a retargetable simulation framework is the ability to automatically decode application binaries of different processor architectures. This section describes the generic instruction decoding technique that is customizable depending on the instruction specifications captured through our generic instruction model.

Algorithm 1 describes how the instruction decoder works [Reshadi et al. 2006]. This algorithm accepts the target program binary and the instruction specification as inputs and generates a source file containing decoded instructions as output. Iterating on the input binary stream, it finds an operation, decodes it using Algorithm 2, and adds the decoded operation to the output source file [Reshadi et al. 2006]. Algorithm 2 also returns the length of the current operation used to determine the beginning of the next operation. Algorithm 2 gets a binary stream and a set of specifications containing operation or microoperation classes. The binary stream is compared to the elements of the specification to find the specification-mask pair that matches the beginning of the stream. The length of the matched mask defines the length of the operation that must be decoded. The types of symbols are determined by comparing their masks with the binary stream. Finally, using the symbol types, all symbols are replaced with their values in the expression part of the corresponding specification. The resulting expression is the behavior of the operation. This behavior and the length of the decoded operation are produced as outputs.

```
Algorithm 1: StaticInstructionDecoder
Inputs: i) Target Program Binary (Application)
       ii) Instruction Specifications (InstSpec)
Output: Decoded Program DecodedOperations
Begin
   Addr = Address of first instruction in Application
   DecodedOperations = {};
   while (Application not processed completely)
     BinStream = Binary stream in Application starting at Addr;
     (Exp, AddrIncrement) = DecodeOperation (BinStream, InstSpec);
     DecodedOperations = DecodedOperations U <Exp, Addr>;
     Addr = Addr + AddrIncrement;
   endwhile;
   return DecodedOperations ;
End;
```

Consider the following SPARC Add operation example and its binary pattern:

```
Add g1, #10, g2 1000-0100 0000-0000 0110-0000 0000-1010
```

Using the specifications described in Section 5.1.1, in the first line of Algorithm 2 the (InegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) pair matches the instruction binary. This means that the *IntegerOps* operation class matches this operation. Next, the values of the four symbols are determined: opcode, dest, src1, src2. Symbol opcode's type is OpTypes, in which the mask pattern of Add matches the operation pattern. So, the value of opcode is Add function. Symbol dest's type is DestType, which is a register type. It is an integer register whose index is bits 25th to 29th (00010), that is, 2. Similarly, the values for the symbols src1 and src2 can be computed. By replacing these values in the expression part of the IntegerOps, the final behavior of the operation would be: g2 = Add(g1, 10); which means g2 = g1 + 10.

```
Algorithm 2: DecodeOperation
Input: Binary Stream (BinStream), Specifications (Spec)
Output: Decoded Expression (Exp), Length (DecodedStreamSize)
Begin
    (OpDesc, OpMask) = findMatchingPair(Spec, BinStream)
    OpBinary = initial part of BinStream whose length is equal to OpMask
    Exp = the expression part of OpDesc;
    foreach pair of (s, T) in the OpDesc
       Find t in T whose mask matches the OpBinary;
       v = ValueOf(t, OpBinary);
       Replace s with v in Exp;
    endfor
    return (Exp , size(OpBinary));
End;
```

5.2 Cycle-Accurate Simulation

Automatic simulator generation for a class of architecture has been addressed in the literature. However, it is difficult to generate simulators for a wide variety of processor and memory architectures including RISC, DSP, VLIW, superscalar, and hybrid. The main bottleneck is the lack of an abstraction (covering a diverse set of architectural features) that permits reuse of the abstraction primitives to compose heterogeneous architectures. In this section, we describe our simulator generation approach based on functional abstraction. Section 5.2.1 presents the functional abstraction needed to capture a wide variety of architectural features. Section 5.2.2 briefly describes ADL-Driven simulator generation using the functional abstraction.

5.2.1 *Functional Abstraction.* In order to understand and characterize the diversity of contemporary architectures, we have surveyed the similarities and differences of each architectural feature in different architecture domains [Mishra et al. 2001]. Broadly speaking, the structure of a processor consists of functional units that are connected using ports, connections, and pipeline latches. Similarly, the structure of a memory subsystem consists of SRAM, DRAM, cache hierarchy, and so on. Although a broad classification makes the architectures look similar, each architecture differs in terms of the algorithm it employs in branch prediction, the way it detects hazards, the way it handles exceptions, and so on. Moreover, each unit has different parameters for different architectures (e.g., number of fetches per cycle, levels of cache, and cache line size). Depending on the architecture, a functional unit may perform the same operation at different stages in the pipeline. For example, read-after-write(RAW) hazard detection followed by operand read happens in the decode unit for some architectures (e.g., DLX [Hennessy and Patterson 2003]), whereas in others these operations are performed in the issue unit (e.g., MIPS R10K). Some architectures even allow operand read in the execution unit (e.g., ARM7). We observed some fundamental differences from this study; the architecture

```
FetchUnit ( # of read/cycle, res-station size, ....)
{
   address = ReadPC();
   instructions = ReadInstMemory(address, n);
   WriteToReservationStation(instructions, n);
   outInst = ReadFromReservationStation(m);
   WriteLatch(decode_latch, outInst);

   pred = QueryPredictor(address);
   if pred {
      nextPC = QueryBTB(address);
      SetPC(nextPC);
   } else
      IncrementPC(x);
}
```

Fig. 13. A fetch unit example.

may use:

—the same functional or memory unit with different parameters;
—the same functionality in different functional or memory units; or
—new architectural features.

The first difference can be eliminated by defining generic functions with appropriate parameters. The second difference can be eliminated by defining generic subfunctions which can be used by different architectures at different stages in the pipeline. The last one is difficult to alleviate since it is new, unless this new functionality can be composed of existing subfunctions. Based on our observations we have defined the necessary generic functions, subfunctions, and computational environments needed to capture a wide variety of processor and memory features. In this section we present functional abstraction by way of illustrative examples. We first explain the functional abstraction needed to capture the structure and behavior of the processor and memory subsystem, then we discuss the issues related to defining generic controller functionality, and finally, we discuss the issues related to handling interrupts and exceptions.

We capture the structure of each functional unit using parameterized functions. For example, a fetch unit functionality contains several parameters, namely, number of operations read per cycle, number of operations written per cycle, reservation station size, branch prediction scheme, number of read ports, number of write ports, etc. These ports are used to connect different units. Figure 13 shows a specific example of a fetch unit described using subfunctions. Each subfunction is defined using appropriate parameters. For example, *ReadInstMemory* reads n operations from an instruction cache using the current PC address (returned by *ReadPC*) and writes them to the reservation station. The fetch unit reads m operations from the reservation station and writes them to the output latch (fetch to decode latch) and uses a BTB-based branch prediction mechanism. We have defined parameterized functions for all functional units present in contemporary programmable architectures, namely, fetch unit, branch prediction unit, decode unit, issue unit, execute unit, completion unit, interrupt handler unit, PC unit, latch, port, connection, and so on.

```
ADD (src1, src2) {
  return (src1 + src2);
}
```

```
MUL (src1, src2) {
  return (src1 * src2);
}
```

```
MAC (src1, src2, src3) {
   return ( ADD( MUL(src1, src2), src3) );
}
```

Fig. 14. Modeling of MAC operation.

We have also defined subfunctions for all the common activities, for example, ReadLatch, WriteLatch, ReadOperand, and so on.

The behavior of a generic processor is captured through the definition of opcodes. Each opcode is defined as a function, with a generic set of parameters, which performs the intended functionality. The parameter list includes source and destination operands as well as necessary control and data type information. We have defined common subfunctions, for example, ADD, SUB, SHIFT, and so on. The opcode functions may use one or more subfunctions, for example, the MAC (multiply and accumulate) uses the two subfunctions (ADD and MUL), as shown in Figure 14.

Each type of memory module, such as SRAM, cache, DRAM, SDRAM, stream buffer, and victim cache, is modeled using a function with appropriate parameters. For example, a typical cache function has many parameters, including cache size, line size, associativity, word size, replacement policy, write policy, and latency. The cache function performs four operations: read, write, replace, and refill. These functions can have parameters for specifying pipelining, parallelism, access modes (normal read, page mode read, and burst read), and so on. Again, each function is composed of subfunctions. For example, an associative cache function can be modeled using a cache function.

A major challenge for functional abstraction of programmable architectures is the modeling of control for a wide range of architectural styles. We define control in both a distributed and centralized manner. Distributed control is transfered through pipeline latches. While an instruction gets decoded, the control information needed to select the operation as well as the source and destination operands are placed in the output latch. These decoded control signals pass through the latches between two pipeline stages unless they become redundant. For example, when the value for *src1* is read, that particular control is not needed any more, and instead the read value will be in the latch. Centralized control is maintained using a generic control table. The number of rows in the table is equal to the number of pipeline stages in the architecture. The number of columns is equal to the maximum number of parallel units present in any pipeline stage. Each entry in the control table corresponds to one particular unit in the architecture. It contains information specific to that unit, for example, busy bit (BB), stall bit (SB), list of children, list of parents, opcodes supported, and so on.

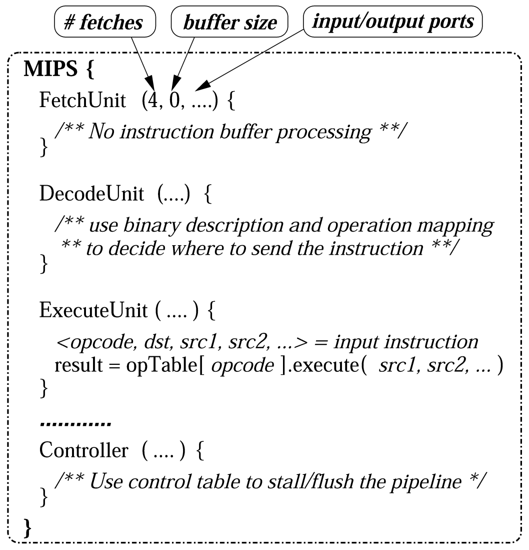
Another major challenge for functional abstraction is the modeling of interrupts and exceptions. We capture each exception using an appropriate subfunction. Opcode-related exceptions (e.g., divide by zero) are captured in the opcode functionality. Functional unit-related exceptions (e.g., illegal slot exception) are captured in functional units. External interrupts (e.g., reset, debug exceptions) are captured in the control unit functionality. A detailed description of functional abstraction for all microarchitectural components is available in Mishra et al. [2001].

5.2.2 *Simulator Generation.* We have developed C++ models for the generic functions and subfunctions described in Section 5.2.1. The development of simulation models is a one-time activity and independent of the architecture. The simulator is generated by composing the abstraction primitives based on the information available in the ADL specification. The simulator generation process consists of three steps. First, the ADL specification is read to gather all the necessary details for the simulator generation. Second, the functionality of each component is composed using generic functions and subfunctions. Finally, the structure of the processor is composed using the structural details. In the remainder of this section we briefly describe the last two steps.

To compose the functionality of each component, all necessary details (such as parameters and functionality) are extracted from the ADL specification. First, we describe how to generate three major components of the processor: instruction decoder, execution unit, and controller, using generic simulation models. Next, we describe how to compose the functionality. A generic instruction decoder uses information regarding individual instruction format and opcode mapping for each functional unit to decode a given instruction. The instruction format information is available in ADL specification. The decoder extracts information regarding opcode and operands from input instruction using the instruction format. The mapping section of the ADL captures the information regarding the mapping of opcodes to functional units. The decoder uses this information to perform/initiate necessary functions (e.g., operand read) and to decide where to send the instruction.

To compose an execution unit it is necessary to instantiate all the opcode functionalities (e.g., ADD, SUB, etc. for an ALU) supported by that execution unit. The execution unit invokes the appropriate opcode functionality for an incoming operation based on a simple table look-up technique, as shown in Figure 15. Also, if the execution unit is supposed to read the operands, the appropriate number of operand read functionalities needs to be instantiated unless the same read functionality can be shared using multiplexers. Similarly, if the execution unit is supposed to write the data back to the register file, the functionality for writing the result needs to be instantiated. The actual implementation of an execute unit might contain many more functionalities, such as read latch, write latch, modify reservation station (if applicable), and so on.

The controller is implemented in two parts. First, it generates a centralized controller (using a generic controller function with appropriate parameters) that maintains the information regarding each functional unit, such as busy, stalled, etc. It also computes hazard information based on the list of instructions currently in the pipeline. Based on these bits and the information



```
MIPS {
  FetchUnit  (4, 0, ....) {
    /** No instruction buffer processing **/
  }

  DecodeUnit  (....)  {
    /** use binary description and operation mapping
     ** to decide where to send the instruction **/
  }

  ExecuteUnit ( .... ) {
    <opcode, dst, src1, src2, ...> = input instruction
    result = opTable[ opcode ].execute(  src1, src2, ... )
  }
  ............
  Controller  ( .... ) {
    /** Use control table to stall/flush the pipeline */
  }
}
```

Fig. 15. An example simulation model for MIPS architecture.

available in the ADL, it stalls/flushes necessary units in the pipeline. Second, a local controller is maintained at each functional unit in the pipeline. This local controller generates certain control signals and sets necessary bits based on the input instruction. For example, the local controller in an execute unit will activate the add operation if the opcode is *add*, or it will set the busy bit in case of a multicycle operation.

It is also necessary to compose the functionality of new instructions (behavior) using the functionality of existing ones. The operation mapping (described in Section 2) is used to generate the functionality for the target (new) instructions using the the functionality of the corresponding generic instructions. The final implementation is generated by instantiating components (e.g., fetch, decode, register files, memories, etc.) with appropriate parameters and connecting them using the information available in the ADL. For example, Figure 15 shows a portion of the simulation model for the MIPS architecture. The generated simulation models combined with the existing simulation kernel create a cycle-accurate structural simulator.

6. DESIGN SPACE EXPLORATION

In this section, we present some sample results of design space exploration of programmable SOCs. We have performed extensive architectural design space exploration by varying different architectural features in the EXPRESSION ADL. We have also performed microarchitectural exploration of the MIPS 4000 processor in three directions: pipeline exploration, instruction-set exploration, and memory exploration [Pasricha et al. 2003]. Pipeline exploration allows the addition (deletion) of new (existing) functional units or pipeline stages. Instruction-Set exploration allows the addition of new instructions or the formation of complex instructions by combining existing instructions. Similarly, memory exploration allows modification of memory hierarchies and cache parameters. The system architect only modifies the ADL description of the architecture, and the software toolkit is automatically generated from the ADL specification using the functional abstraction approach. The public release of the retargetable simulation and exploration framework is available from http://www.cecs.uci.edu/˜express. This release also supports a graphical user interface (GUI). The architecture can be described (or modified) using the GUI. The ADL specification and the software toolkit are automatically generated from the graphical description. In the remainder of this section, we present sample experimental results demonstrating the need for and usefulness of the compiler-in-the-loop design space exploration of rISA designs which was discussed in Section 4.

6.1 Instruction-Set Exploration (rISA Design Space)

The experimental results of the compiler-in-the-loop design space exploration of rISA designs is divided into two main parts. First, we demonstrate the goodness of our rISA compiler. We show that, as compared to existing rISA compilers, our instruction-level register pressure-sensitive compiler can consistently achieve superior code compression over a set of benchmarks. In the second part, we develop rISA designs and demonstrate that the code compression achieved by using different rISA designs is very sensitive to the rISA design itself. Depending on the specific rISA design chosen, there can be a difference of up to 2X in the code compression achievable. This clearly demonstrates the need to very carefully select the appropriate rISA design, and shows that a compiler-in-the-loop exploration greatly helps the designer choose the best rISA configuration.

6.1.1 *rISA Compiler Comparison.* For the baseline experiments, we compiled the applications using GCC for MIPS32 ISA. Then we compiled the same applications using GCC for MIPS32/16 ISA. We performed both compilations using -Os flags with the GCC to enable all the code size optimizations. The percentage code compression achieved by GCC for MIPS16 was computed and is represented by the light bars in Figure 16 taken from Halambi et al. [2002]. The MIPS32 code generated by GCC was compiled again using the register pressure-based heuristic in our EXPRESS compiler. The percentage code compression achieved by EXPRESS was measured and is plotted as dark bars in Figure 16.

Figure 16 shows that the register pressure-based heuristic performs consistently better than GCC and successfully prevents code size inflation. GCC achieves, on average, a 15% code size reduction, while EXPRESS achieved an average of 22% code size reduction. We simulated the code generated by EXPRESS on a variant of the MIPS R4000 processor that was augmented with the rISA MIPS16 instruction set. The memory subsystem was modeled with no caches and a single-cycle main memory. The performance of MIPS16 code is, on average, 6% lower than that of MIPS32 code, with the worst case being 24% lower. Thus, our technique was able to reduce the code size using rISA with a minimal impact on performance.

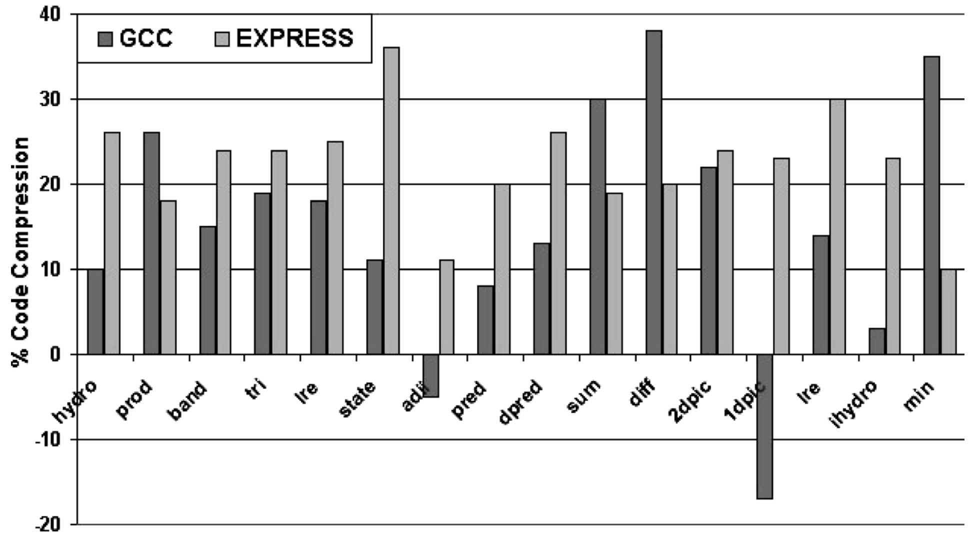


Fig. 16. Percentage code compressions achieved by GCC and EXPRESS for MIPS32/16.

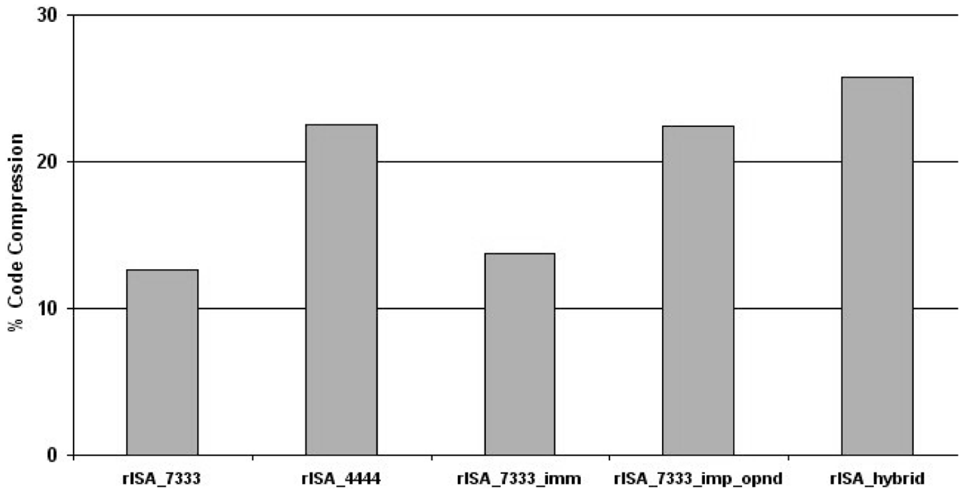


Fig. 17. Code size reduction for various rISA architectures.

6.1.2 *Sensitivity to rISA Designs.* Owing to various design constraints on rISA, the code compression achieved by using an rISA is very sensitive to the rISA chosen. The rISA design space is huge and several instruction-set idiosyncrasies make it very difficult to characterize. To show the variation of code compression achieved with rISA, we took a practical approach. We systematically designed several rISAs, and studied the code compression achieved by them. Figure 17 taken from Halambi et al. [2002] plots the code compression achieved by each rISA design. We started with the extreme rISA designs, *rISA_7333* and *rISA_4444*, and gradually improved upon them.

rISA_7333. The first rISA design is (*rISA_7333*). In this rISA, the operand is a 7-bit field, and each operand is encoded in 3-bits. Thus there can be $2^7 = 128$ instructions in this rISA, but each instruction can access only 8 registers. Furthermore, they can support unsigned immediate values from 0 to 7. However, instructions that have 2 operands (like move) have 5-bit operands. Thus, they can access all 32 registers. Owing to the uniformity in the instruction format, the translation unit is very simple for this rISA design. The leftmost bar in Figure 17 plots the average code compression achieved when we used the *rISA_7333* design on all our benchmarks. On average, *rISA_7333* achieved 12% code compression. The *rISA_7333* is unable to achieve good code compressions for applications that have high register pressure, for example, *adii*, and those with large immediate values. In such cases, the compiler heuristic decides not to rISAize large portions of the application in order to avoid code size increase due to extra spill/reload and immediate extend instructions.

rISA_4444. The *rISA_4444* is the instruction set on the other end of the rISA design spectrum. It provides only 4-bits to encode the opcode, but has 4-bits to specify each operand as well. Thus, there are $2^4 = 16$ instructions in *rISA_4444*, but each instruction can access $2^4 = 16$ registers. We profiled the applications and incorporated the 16 most frequently occurring instructions in this rISA. The second bar from the left in Figure 17 shows that the register pressure problem is mitigated in the *rISA_4444* design. It achieves better code size reduction for benchmarks that have high register pressure, but performs badly on some of the benchmarks because of its inability to convert all the normal instructions into rISA instructions. An *rISA_4444* achieves about 22% improvement on the normal instruction set.

rISA_7333_imm. We now attack the second problem in *rISA_7333*—small immediate values. For instructions that have immediate values, we decreased the size of opcode, and used the bits to accommodate as large an immediate value as possible. This design point is called *rISA_7333_imm*. Because of the nonuniformity in the size of the opcode field, the translation logic is complex for such an rISA design. The middle bar in Figure 17 shows that *rISA_7333_imm* achieves slightly better code compressions as compared to the first design point, since it has large immediate fields while having access to the same set of registers. An *rISA_7333_imm* achieves about 14% improvement on the normal instruction set.

rISA_7333_imp_opnd. Another useful optimization that can be performed to save precious bit space is to encode instructions with the same operands using a different opcode. For example, suppose we have a normal instruction *add R1 R2 R2*, and suppose we have an rISA instruction of the format *rISA_add1 R1 R2 R3*. The normal instruction can be encoded into this rISA instruction by setting the two source operands the same (equal to $R2$). However, having a separate rISA instruction format of the form *rISA_add2 R1 R2* to encode such instructions may be very useful. This new rISA instruction format has fewer operands, and will therefore require fewer instruction bits. The extra bits can be used in two ways: first, directly by providing increased register file access to the remaining operands, and second, indirectly. Since this instruction can afford a longer opcode, another instruction with tighter constraints on the opcode field (e.g., an instruction with immediate operands) can switch opcode with this instruction.

We employed the implied operands feature in *rISA_7333* and generated our fourth rISA design, *rISA_7333_imp_opnd*. This rISA design matches the MIPS16 rISA. The second bar from the right in Figure 17 represents the code compression achieved by the *rISA_7333_imp_opnd*. The *rISA_7333_imp_opnd* design achieves about the same code size improvement as the *rISA_4444* design point; it achieves about 22% code compression over the normal instruction set.

rISA_hybrid. Our final rISA design point is *rISA_hybrid*. This is a custom ISA for each benchmark. All the previous techniques, that is, long immediate values, implied operands, etc., are employed to define the custom rISA for each benchmark. In this rISA design instructions can have variable register accessibilities. Complex instructions with operands having different register set accessibilities are also supported. The register set accessible by operands varies from 4 to 32 registers. We profiled the applications and manually (heuristically) determined the combinations of operand bit-width sizes that provided the best code size reduction. The immediate field was also customized to gain best code size reduction. The *rISA_hybrid* achieves the best code size reduction since it is customized for the application set. The rightmost bar in Figure 17 shows that *rISA_Hybrid* achieves about 26% overall improvement on the normal instruction set. The code compression is consistent across all benchmarks.

In summary, our experimental results for rISA-based code compression show that the effects of different rISA formats are highly sensitive to the application characteristics: the choice of a rISA format for different applications can result in up to a 2X increase in code compression. However, the system designer critically needs a compiler-in-the-loop approach to evaluate, tune, and design the rISA instruction set so as to achieve the desired system constraints of performance, code size, and energy consumption.

7. CONCLUSIONS

The complexity of programmable architectures (consisting of processor cores, coprocessors, and memories) is increasing at an exponential rate due to the combined effects of advances in technology as well as demands from increasingly complex application programs in embedded systems. The choice of programmable architectures plays an important role in SOC design due to its impact on the overall cost, power, and performance. A major challenge for an architect is to find the best possible programmable architecture for a given set of application programs and various design constraints. Due to the increasing complexity of programmable architectures, the number of design alternatives is extremely large. Furthermore, shrinking time-to-market constraints make it impractical to explore all the alternatives without using an automated exploration framework. This article presented an architecture description language (ADL)-Driven exploration methodology that is capable of accurately capturing a wide variety of programmable architectures and generating efficient software toolkits, including compilers and simulators.

ADLs have been successfully used in academic research as well as in industry for embedded processor development. The early ADLs were either structure-oriented (MIMOLA), or behavior-oriented (nML, ISDL). As a result, each class of ADLs is suitable for specific tasks. For example, structure-oriented ADLs are suitable for hardware synthesis, and unfit for compiler generation. Similarly, behavior-oriented ADLs are appropriate for generating compilers and simulators for instruction-set architectures, but unsuited for generating cycle-accurate simulators or hardware implementations of the architecture. However, a behavioral ADL can be modified to perform the task of a structural ADL (and vice versa). For example, nML is extended by Target Compiler Technologies to perform hardware synthesis and test generation (http://www.retarget.com). The recent ADLs (LISA, EXPRESSION) adopt the mixed approach where the language captures both the structure and the behavior of the architecture. 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 (in terms of area, power, and performance) tools and hardware prototypes. 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 tools/prototypes compared to domain-specific or task-specific ADLs.

This article presented the four important steps in an ADL-Driven exploration methodology: architecture specification, validation of specification, retargetable software toolkit generation, and design space exploration. The first step is to capture the programmable architecture using an ADL. The next step is to verify the specification to ensure the correctness of the specified architecture. The validated specification is used to generate a retargetable software toolkit, including a compiler and a simulator. This article presented sample experiments to illustrate the use of an ADL-Driven architectural exploration methodology for the exploration of reduced bit-width instruction-set architectures (rISA) on the MIPS platform. Our experimental results demonstrated the need for and utility of the compiler-in-the-loop exploration methodology driven by an ADL specification. The ADL specification can also be used for rapid prototyping [Schliebusch et al. 2002; Mishra et al. 2003a; Leupers and Marwedel 1998], test generation (http://www.retarget.com), [Leupers and Marwedel 1998; Mishra and Dutt 2004b] and functional verification of programmable architectures [Mishra and Dutt 2005].

As SOCs evolve in complexity to encompass high degrees of multiprocessing coupled with heterogeneous functionality (e.g., MEMS and mixed signal devices) and new on-chip interconnection paradigms (e.g., networks-on-chip), the next generation of multiprocessor SOCs (MPSOCs) will similarly require a language-driven methodology for the evaluation, validation, exploration, and codesign of such platforms.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions of Professors Alex Nicolau, Mehrdad Reshadi, Ashok Halambi, Dr. Peter Grun, Dr. Partha Biswas, Dr. Mahesh Mamidipaka, and Sudeep Pasricha.

REFERENCES

Advanced RISC Machines Ltd. *An Introduction to Thumb*. Advanced RISC Machines Ltd.

http://www.arccores.com. *ARCtangent-A5 microprocessor Technical Manual*.

BARBACCI, M. R. 1981. Instruction set processor specifications (ISPS): The notation and its applications. *IEEE Trans. Comput. 30*, 1, 24–40.

BRIGGS, P., COOPER, K., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

CLEMENTS, P. C. 1996. A survey of architecture description languages. In *Proceedings of the International Workshop on Software Specification and Design (IWSSD)*, 16–25.

http://www.coware.com. *COWare LISATek*.

FREERICKS, M. 1993. The nML machine description formalism. Tech. Rep. TR SM-IMP/DIST/08, TU Berlin, Computer Science Department.

HADJIYIANNIS, G., RUSSO, P., AND DEVADAS, S. 1999. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of the Design Automation Conference (DAC)*, 927–932.

HADJIYIANNIS, G., HANONO, S., AND DEVADAS, S. 1997. ISDL: An instruction set description language for retargetability. In *Proceedings of the Design Automation Conference (DAC)*, 299–302.

HALAMBI, A., SHRIVASTAVA, A. DUTT, N., AND NICOLAU, A. 2001. A customizable compiler framework for embedded systems. In *Proceedings of the Software and Compilers for Embedded Systems (SCOPES) Conference*.

HALAMBI, A., SHRIVASTAVA, A., BISWAS, P., DUTT, N., AND NICOLAU, A. 2002. An efficient compiler technique for code size reduction using reduced bit-width isas. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*.

HALAMBI, A., GRUN, P., GANESH, V., KHARE, A., DUTT, N., AND NICOLAU A. 1999. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*. 485–490.

HENNESSY, J. AND PATTERSON, D. 2003. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Fransisco, Calif.

KISSELL, K. 1997. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group.

LEUPERS, R. AND MARWEDEL, P. 1998. Retargetable code generation based on structural processor descriptions. *Des. Autom. Embedded Syst. 3*, 1, 75–108.

LSI LOGIC. *TinyRISC LR4102 Microprocessor Technical Manual*. LSI LOGIC.

MDES User Manual. 1997. *http://www.trimaran.org*. The MDES User Manual.

MISHRA, P., ASTROM, J., DUTT, N., AND NICOLAU, A. 2001. Functional abstraction of programmable embedded systems. Tech. Rep. UCI-ICS 01-04, University of California, Irvine, Jan.

MISHRA, P., TOMIYAMA, H., DUTT, N., AND NICOLAU, A. 2002. Automatic verification of in-order execution in microprocessors with fragmented pipelines and multicycle functional units. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 36–43.

MISHRA, P., KEJARIWAL, A., AND DUTT, N. 2003a. Rapid exploration of pipelined processors through automatic generation of synthesizable RTL models. In *Proceedings of the Rapid System Prototyping (RSP) Conference*, 226–232.

MISHRA, P., DUTT, N., AND TOMIYAMA, H. 2003b. Towards automatic validation of dynamic behavior in pipelined processor specifications. *Des. Autom. Embedded Syst. 8*, 2–3 (June-Sept.), 249–265.

MISHRA, P. AND DUTT, N. 2004a. Automatic modeling and validation of pipeline specifications. *ACM Trans. Embedded Comput. Syst. 3*, 1, 114–139.

MISHRA, P. AND DUTT, N. 2004b. Graph-Based functional test program generation for pipelined processors. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*, 182–187.

MISHRA, P. AND DUTT, N. 2005a. Architecture description languages for programmable embedded systems. *IEE Proceedings Comput. Digital Techniques 152*, 3 (May), 285–297.

MISHRA, P. AND DUTT, N. 2005. *Functional Verification of Programmable Embedded Architectures: A Top-Down Approach*. Springer Verlag, New York.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, Calif.

Park, S., Shrivastava, A., Earlie, E., Dutt, N., Nicolau, A., and Paek, Y. 2006. Automatic generation of operation tables for fast exploration of bypasses in embedded processors. In *Proceedings of the Design Automation and Test in Europe (DATE) Conference*.

Pasricha, S., Biswas, P., Mishra, P., Shrivastava, A., Mandal, A., Dutt, N., and Nicolau, A. 2003. A framework for GUI-Driven design space exploration of a MIPS4K-like processor. Tech. Rep. CECS 03-17, University of California, Irvine.

Pees, S., Hoffmann, A., and Meyr, H. 2000. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Trans. Des. Autom. Electronic Syst. 5*, 4, 815–834.

Qin, W. and Malik, S. 2002. Architecture description languages for retargetable compilation. In *the Compiler Design Handbook: Optimizations & Machine Code Generation*. CRC Press, Boca Raton, Fla.

Reshadi, M., Bansal, N., Mishra, P., and Dutt, N. 2003a. An efficient retargetable framework for instruction-set simulation. In *Proceedings of the International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 13–18.

Reshadi, M., Mishra, P., and Dutt, N. 2003b. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *Proceedings of the Design Automation Conference (DAC)*, 758–763.

Reshadi, M., Mishra, P., and Dutt, N. 2006. A retargetable framework for instruction-set architecture simulation. To appear in *ACM Trans. Embedded Comput. Syst.*

Schliebusch, O., Hoffmann, A., Nohl, A., Braun, G., and Meyr, H. 2002. Architecture implementation using the machine description language LISA. In *Proceedings of the Asia South Pacific Design Automation Conference (ASPDAC)/International Conference on VLSI Design*, 239–244.

Shrivastava, A. and Dutt, N. 2004. Energy efficient code generation exploiting reduced bit-width instruction set architectures (risa). In *Proceedings of the Conference on Asia South Pacific Design Automation*. IEEE Press, Piscataway, N.J. 475–477.

Shrivastava, A., Earlie, E., Dutt, N., and Nicolau, A. 2004. Operation tables for scheduling in the presence of incomplete bypassing. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM Press, New York, 194–199.

Shrivastava, A., Dutt, N., Nicolau, A., and Earlie, E. 2005. Pbexplore: A framework for compiler-in-the-loop exploration of partial bypassing in embedded processors. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*. IEEE Computer Society, Washington, D.C. 1264–1269.

Shrivastava, A., Biswas, P., Halambi, A., Dutt, N., and Nicolau, A. 2006. Compilation framework for code size reduction using reduced bit-width isas. *ACM Trans. Des. Autom. Electronic Syst.*

Siska, C. 1998. A processor description language supporting retargetable multi-pipeline DSP program development tools. In *Proceedings of the International Symposium on System Synthesis (ISSS)*, 31–36.

http://www.sparc.org. The *SPARC Architecture Manual*.

ST100 DSP-MCU Architecture. http://www.st.com. The ST100 DSP-MCU Architecture.

http://www.retarget.com. *Target Compiler Technologies.*

Tensilica Inc. http://www.tensilica.com. Tensilica Inc.

Tomiyama, H., Halambi, A., Grun, P., Dutt, N., and Nicolau, A. 1999. Architecture description languages for systems-on-chip design. In *Proceedings of the Asia Pacific Conference on Chip Design Language*, 109–116.

http://www.ics.uci.edu/˜express. *Exploration framework using EXPRESSION ADL*.

Zivojnovic, V., Pees, S., and Meyr, H. 1996. LISA—Machine description language and generic machine model for HW/SW co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing*, 127–136.

Received February 2006; accepted May 2006