MASTERARBEIT

Automatic Generation of Interpreting Instruction Set Simulators

ausgeführt am Institut für
Computersprachen
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

durch
ANDREAS FELNHOFER
Hengstberg 23
3372 Blindenmarkt

Wien, am 30. September 2008
Abstract

Generated design tools can make the design cycle of application specific embedded processors more efficient and predictions about runtime behaviour of a hardware/software co-design more reliable. For example simulation results drawn from generated simulators can guide a systems architect through the numerous alternatives and tradeoffs in hardware designs.

Architecture description languages are used to formalise a processor design on a medium level of abstraction - such a description can be used as an input for design tool generators. The xADL language follows a structure-centric approach and allows a wide range of tools to be generated (e.g. simulators, compilers, binary utils). It has been developed at the TU Vienna, Institute of Computer Languages, together with a framework pre-analysing the input and allowing different modules to process it - which is the starting point of this work.

A method to translate an xADL description to a cycle-accurate simulator including an instruction decoder is presented and its implementation in form of a simulator generator module is described. Furthermore this work introduces a technique to generate instruction decoders necessary for the generated simulators.
Kurzfassung

Automatisch generierte Entwicklungswerkzeuge können einen wesentlichen Beitrag leisten, den Entwurfszyklus von anwendungsspezifischen Embedded Prozessoren zu verkürzen und zuverlässigere Abschätzungen über das Laufzeitverhalten von Hardware/Software-Systemen zu treffen. Ergebnisse aus Testläufen eines generierten Simulators zum Beispiel sind ein probates Mittel um aus den unzähligen Alternativen bei der Hardware Entwicklung passende Kompromisse zu wählen.


Es wird eine Methode vorgestellt, die es erlaubt, xADL Beschreibungen in zyklengenaue Simulatoren zu übersetzen und eine konkrete Implementierung in Form eines (nicht zyklengenauen) Generator Moduls beschrieben. Des Weiteren präsentiert die vorliegende Arbeit eine Technik um Instruktionsdecoder, wie sie für Simulatoren üblicherweise benötigt werden, zu generieren.
Contents

Abstract ii
Kurzfassung iii
Introduction 7

1 Computer Architectures 9
  1.1 Instruction Set Architectures 10
     1.1.1 Categorisation 10
     1.1.2 Operations 11
     1.1.3 Operands 12
     1.1.4 Memory Access 12
  1.2 Computer Micro Architecture 13
     1.2.1 Pipelining 14
     1.2.2 Stalling 14
     1.2.3 Forwarding 15
     1.2.4 Caching 15
     1.2.5 Branch Prediction 15
     1.2.6 Advanced Pipelining 16

2 Architecture Description Languages 18
  2.1 Classification 18
  2.2 Related Work 19
     2.2.1 MIMOLA 19
     2.2.2 ISDL 20
     2.2.3 Expression 20
     2.2.4 LISA 21
     2.2.5 nML 21
     2.2.6 MADL 21
     2.2.7 PD-XML 21
     2.2.8 ArchC 22
     2.2.9 Generic Netlists 22
     2.2.10 Portable Compiled Instruction-set Simulators 22
## CONTENTS

2.3 xADL .................................................. 23
   2.3.1 Syntax .......................................... 24
   2.3.2 Instruction Set .................................. 27

3 Simulator Generator ........................................... 28
   3.1 Simulator Framework ................................ 29
      3.1.1 Simulation Model ................................ 29
      3.1.2 Architectural State .............................. 30
      3.1.3 Pipeline Organisation ......................... 30
      3.1.4 Pipeline Simulation ............................ 31
   3.2 System Calls ....................................... 32
   3.3 Binary File Loading .................................. 32
   3.4 Internal Description ................................ 33
      3.4.1 Memory-Elements .............................. 35
      3.4.2 Call Sequence ................................. 35
      3.4.3 Condition Tree ................................ 36
      3.4.4 Building up Internal Description ............. 36

4 Simulator Generator Implementation .......................... 37
   4.1 The Web ........................................... 38
      4.1.1 Components .................................. 38
      4.1.2 Built-in Calls ................................ 39
      4.1.3 Instructions .................................. 39
   4.2 Basic Principles .................................... 39
      4.2.1 Pipeline Stage Extraction ..................... 41
      4.2.2 Component and Connection Translation ....... 41
      4.2.3 Call Environment ................................ 42
      4.2.4 Memory Element Handling .................... 43
      4.2.5 Operand Handling ................................ 44
      4.2.6 Conditional Move Conversion ................. 44
   4.3 Hazard Translation ................................ 46
      4.3.1 Resolving Forwards ............................ 46
      4.3.2 Concurrent Memory Access .................... 49
      4.3.3 Concurrent Register Access ................... 49
      4.3.4 Stalling ....................................... 50
   4.4 Further Translations ................................ 51
      4.4.1 VLIW Pipeline Stages ......................... 51
      4.4.2 Fetch/Decode .................................. 52
      4.4.3 Symbolic Values ................................ 52

5 The Interpreting Simulator ...................................... 53
   5.1 Generator Backend .................................. 53
      5.1.1 Global Memory Element Pool .................. 54
## CONTENTS

5.1.2 Operand Memory Element Pool ........................................ 54  
5.1.3 Pipestage-Functions .................................................... 54  
5.1.4 Further Backend Tasks ................................................ 55  
5.2 Instruction Decoder ....................................................... 55  
  5.2.1 Related Work .......................................................... 55  
  5.2.2 xADL Binary Encoding Extension .................................... 56  
  5.2.3 VLIW Binary Encoding ............................................... 57  
  5.2.4 Decoder Generation .................................................. 57  
6 Evaluation ............................................................................ 60  
  6.1 Generator Results .......................................................... 60  
  6.2 Simulation Technique and Methodology ................................. 61  
    6.2.1 Simulator Validation and Accuracy ................................. 61  
    6.2.2 Processor and Memory Parameter Values ......................... 62  
    6.2.3 Selected Benchmarks and Input Sets ............................... 62  
    6.2.4 Simulation ............................................................. 62  
  6.3 Simulator Results .......................................................... 64  
    6.3.1 Simulation Speed ...................................................... 64  
    6.3.2 Simulation Time Breakdown ....................................... 65  
Conclusion .............................................................................. 67  
Bibliography ............................................................................ 70
Introduction

The Institute of Computer Languages at the TU Vienna has initiated the development of a generic architecture description language called xADL. It originates in the need for optimising retargetable compilers for application-specific embedded processors and revealed to be generic enough for the automatic generation of a wide range of hard- and software components like processor simulators or even hardware descriptions (VHDL). Besides the definition of the xADL language, an extensible framework is provided that comprises a variety of modules. These modules analyse a described architecture and generate specific parts of its toolchain.

In the course of two master theses a simulator generator module was implemented. Besides the development of a generator, producing retargetable simulators with instruction decoders, techniques of dynamic binary translation were applied to speed up these simulators. The design and implementation of the simulator generator and its optimisations were done in close collaboration of the authors. As two separate theses were intended the presentation of the overall project is divided into two individual parts: While one explains all necessary steps needed to translate an xADL description to a full-blown interpreting simulator, the other one focuses on applying dynamic binary translation in the context of these retargetable simulators.

As the fundamentals of the two theses have the same origin, the first chapters (including parts of this introduction) were authored in close cooperation and are identical in both works. At the beginning of the shared part various aspects of computer architectures to be considered for description languages and simulations are summarised (with a focus on embedded systems). A choice of description languages is categorised and compared to the philosophy of xADL. Its syntax and semantic are briefly defined and language constructs are outlined. The last shared chapter describes the simulation framework (those parts of the simulator not being generated) and roughly explains an internal description used as the basis for the two different parts of generated simulators: the interpreter and the dynamic binary translator.

First, the generators internal structure as well as the translations necessary to build up its so-called internal description are presented. Roughly spoken this description comprises of data structures that model sequential code and data storages and it can fairly easy be
converted to C++ code. As a prerequisite, xADL language constructs have to be converted to an object oriented representation (called “web”) by the xADL framework. The transformation of the web to the generator modules internal description is the sophisticated process of transferring a structural description to a (quasi) executable one. It must respect architectural aspects like hazards and is covered by the main chapter of this work titled “Simulator Generator Implementation”.

The final conversion of the internal description to compilable C++ code and the process of generating instruction decoders form the second topic of my work and are presented in Chapter 5. Finally, the generated simulators are evaluated: After the simulation process is described, the accuracy and performance of two concrete generated interpreting simulators are compared to commercial products.

This work is supported in part by ON DEMAND Microelectronics and the Christian Doppler Forschungsgesellschaft.
Chapter 1

Computer Architectures

It may be helpful to have definitions at hand when introducing abstract concepts like computer architecture. But as both terms “computer” and “architecture” cover wide areas in culture, science and everyday life such an exact definition can hardly be stated without some aspects getting out of sight.

To approach computer architectures let us first ask for the purpose of architecture in terms of planning and constructing buildings. Buildings serve fundamental and practical human needs like protection against natural environment as well as aesthetical ones. In practice there are a variety of subtle needs, goals and constraints a building has to meet. Without getting too philosophical this view clearly focuses on material aspects - the intended end product.

A similar view can be put on micro processor architecture. For one of the biggest sectors of the computer market - the desktop computer - there is a demand for the highest performance available per cost unit. This manifests in the end product: Small feature sizes provide maximal transistor counts, allowing expensive (in terms of transistors) performance optimisations at reasonable price.

When looking at a computer architecture, one can ask, what the reasons for design decisions might have been and find explanations for them - but immediately the question arises if one can think of a better implementation. This leads to another view: Architecture can be seen as the translation of abstract ideas to a concrete plan. A plan that most likely (but sometimes only hopefully) covers all needs and respects all constraints. Chapter 2 will cover architecture description languages that account for this view and allow to formalise, to plan computer architectures.
1.1 Instruction Set Architectures

A look at the computer’s history reveals that computer architecture first was a synonym for instruction set architecture. The instruction set architecture defines the number and types of operations, registers and memory addressing modes to be covered by the instruction set. The need for convenient assembler programming, maximal instruction execution frequency and minimal instruction count was initially seen as a problem of designing optimal instructions. It revealed that rapidly growing register budgets allowed on-chip dynamic translation of instruction sets - the structural implementation of computer (micro)architectures began to play the key role.

Nevertheless the instruction set architecture is important, as it is the interface to the micro architecture - still an instruction set may be favoured over another depending on how computer resources are accessed. Scientific computing may favour instruction sets with special operations while for complicated string operations rich addressing modes and conversion operations may be interesting. Recent instruction sets feature subsets that can be dynamically switched between (e.g. arm thumb) or high parallelism to be solved at compile time (VLIW architectures).

The following sections are a rough overview and mainly summarise [Hennessy and Patterson, 2007].

1.1.1 Categorisation

Instruction set architectures differ in how operands and results are treated - a first categorisation of instruction sets pays attention to these differences. Typically sources for operands and destinations for results are organised as register files, linear memories or stacks.

Stack: All operands are taken from the top of a memory stack. The result is pushed back on top again.

Accumulator: One special register (the accumulator) is implicitly taken as one operand. The result of an operation is stored back to the accumulator again.

Memory/Register: Operands of operations can be either taken from memory or a register file. The result may be stored back to a register or a memory location.

Register/Register Load/Store: Operands must be taken from register files, the result must be stored back to a register. Access to memory is realised solely by special load/store instructions.
1.1.2 Operations

The main purpose of an instruction set is to define the operations a micro architecture can perform per instruction cycle. One categorisation divides instructions into data manipulation instructions and control flow instructions.

Finer differentiation of data manipulation instructions divide them further into arithmetic and logical, data transfer, floating point, decimal and string manipulation instructions. Every field of operation may favour some special purpose operations like multiply-add for signal processing, powerful bit manipulation for control systems or high precision floating point for scientific calculations.

It is worth to take a closer look at control flow instructions, as they are some of the most frequently executed instructions, terminate basic blocks and need special handling in simulators that use binary translation. In general they can be divided into three categories (compare with [Hennessy and Patterson, 2007]):

**Conditional Branches:** Change program flow under certain conditions

**Jumps:** Unconditionally change program flow

**Calls and Returns:** Change program flow, save/restore the program counter

Control flow instructions (called jumps for simplification) must define how the program counter is modified. Either it is directly set to an absolute value (register value, value in memory, etc.) or it is modified by adding a displacement, which is called PC-relative. PC-relative addressing saves bits for short distance jumps and makes code position-independent. Absolute jumps are used for constant jump targets like long distance calls but also when jumping to calculated addresses (typically called computed jump) like dynamically loaded library functions, virtual functions and branches of a switch statement. Calculated jumps play a key role in simulators as they cannot be easily evaluated before runtime. It should be noticed, that the return instruction is a special form of a calculated jump.

Theoretically every state within the microarchitecture is a candidate to be used as a condition for branches. In practice either a single register value, the relation between two registers, a single flag or the logical combination of flags determine if a branch is taken or not.

Some instruction set architectures (like ARM) define conditional execution not exclusively for branch instructions. Another extension to conditional execution are speculative instructions. These can be used to give the microarchitecture hints, if a branch is likely to be taken or to prefetch data (speculative loads). Instruction set architectures that are designed for low power consumption introduce even more special techniques. Some of
these are: loads and stores that bypass the tag memory of a cache (a hit must be statically known), a register file hierarchy, energy-reduced instruction alternatives and explicit exception state management [Asanovic, 2000].

1.1.3 Operands

Operations can be seen as functions taking one or more input values producing (after a finite amount of microarchitecture cycles) one or more output values. The categorisation of different sources and destinations will be fuzzy, as theoretically every architectural state may be a source operand or a destination for a result:

- Registers, often specified by their index within a register file
- Internal state, often defined by a name (e.g. flags)
- Constant values, immediate values (also called constants and immediates)
- Memory objects, specified by their address

Section 1.1.4 describes handling of memory objects. Subsequently some aspects of register file access will (not exhaustively) be mentioned. First, registers may be divided into several banks. Each bank may be used by a subset of instructions or banks may act as shadow registers that can be dynamically switched. Partitioning to form register file hierarchies pays attention to the principle of locality (like caches do). Finally subregister naming allows instructions to access parts of a register by a special name (e.g. within the x86 instruction set).

1.1.4 Memory Access

An instruction set architecture must define how memory is accessed. Theoretically every method that brings bits onto the data/address bus of a memory is relevant. But with description languages in mind only the most common and important aspects of memory accesses should be summarised here.

Byte order is a first aspect differing from machine to machine. Byte order is relevant whenever multiple bytes are read/written to form larger entities (e.g. 32 Bit words). If a 4 Byte word is written and read again, the internal order of bytes is irrelevant, but if the same word is read with single byte accesses the order does matter. More precise: It defines how small portions of memory are aligned within larger ones.

The two common byte orders are called little endian and big endian. Little endian puts the least significant byte to the lowest byte address and the following ones to ascending
addresses, while big endian places bytes exactly reverse. Order can also be relevant for smaller portions of memory and even single bits, but smaller than byte sized memory accesses are uncommon, single bits are usually masked out of larger entities.

Data transfer sizes within an instruction set are typically the size of the data bus and multiples/fractions of it. The MIPS architecture for example has a 32 Bit wide data bus and allows 8, 16, 32 and for special cases 64 Bit wide objects to be accessed within one instruction. An access to an object of size $S$ at address $A$ is called aligned if $A \mod S = 0$. Some architectures do not allow unaligned accesses, as they might impose multiple accesses to memory.

Another aspect of memory usage within the instruction set is the calculation of memory addresses. Typically the address used by an instruction is composed of a base address and a displacement (which is added to the base). The base typically is a register value (represented in the instruction by a register index) and the displacement may be a register or an immediate value. Techniques like memory management or usage of multiple memories lead to more complicated memory addressing.

Furthermore a memory access may not only fetch data but also increase registers (by a displacement or a constant), make physical out of virtual addresses, throw exceptions caused by violation of permissions or ranges and finally may reorder bytes. Thus memory accesses are not trivial to simulate, which is also true for micro architecture in general as methods to overcome the gap between memory bandwidth and latency (e.g. caches) cause high complexity.

1.2 Computer Micro Architecture

The micro architecture of a processor deals with the processors internal structure. It mainly provides a correct implementation of the instruction set making technical details like access to a cache, branch prediction or address calculations transparent. Of course these details are not fully hidden, they manifest (for example) in bounds for instruction execution times.

For a simulator this means that the more accuracy is required, the more aspects of the micro architecture have to be simulated. Without going into too much details this chapter presents those structures and methods used by micro architectures influencing cycle counts - targeting structures to be simulated by cycle accurate simulators. The intended use of the simulator for embedded systems further narrows down the topics.
1.2.1 Pipelining

“Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs” [Hennessy and Patterson, 2007].

Each instruction is divided into operations to be handled at each pipeline stage. For a given instruction set this division is not immediately clear. The longest delay of a stage determines the speed of the whole pipeline and pipeline hazards may occur. On the other hand pipeline design may influence instruction set decisions.

Hazards are the danger of influencing the instruction set in a way that was not intended: the danger of producing wrong results. They result from the nature of pipelined execution and are typically divided into three categories.

**Structural Hazards:** A resource is needed in \( n \) different stages simultaneously - only less than \( n \) accesses are possible.

**Data Hazards:** Two instructions are in the pipeline. Read after Write (RAW) also called true dependency: the result of one instruction is needed by another one in the pipeline. Write after Write (WAW) and Write after Read (WAR), also called anti dependencies: writebacks to the register file may be out of order and cause wrong register file content (at some time).

**Control Hazards:** The program counter is changed - some instructions not being intended to get executed are in the pipeline.

One way to overcome pipeline hazards without introducing new hardware structures is to simply forbid instruction sequences causing them - the compiler must rearrange instructions or insert instructions that do not modify the architectural state (no-operations or nops) to overcome hazards. Below other methods are presented that do have impact to hardware structures in micro architectures, each addressing one or more types of hazards.

1.2.2 Stalling

A prominent structural hazard occurs in Von Neumann architectures: the simultaneous access of the memory by the instruction fetch unit and some other unit needing an operand from memory. A solution is to stall the pipeline: The fetch stage neither touches the memory nor increases the program counter during that cycle - it produces a bubble (nop instruction) instead. The bubble resolves the conflict but consumes one extra cycle (the real fetch is delayed).
More general: If the hazard caused by two stages with numbers $i$ and $j$ (let $i < j$) is to be resolved by stalling, stage $i$ delays its conflicting operation until the conflict is resolved. Every delayed cycle a nop is passed to the successor stage and all stages $< i$ are blocked.

### 1.2.3 Forwarding

Data hazards may also be solved by stalling, but it is more efficient (in terms of cycles needed) to directly connect the output of one unit producing the result to the input of another unit needing it as an operand. For example the result of the ALU can be forwarded to both inputs of the ALU itself. Two criteria must apply to make use of the forwarded value: The destination register index for that value must be the same as for the input value and the forwarded value must be valid for that cycle. Thus only the needed register value (no future or former value) is forwarded.

### 1.2.4 Caching

A cache together with stalling can also resolve structural hazards. It is a special form of memory duplication: It holds a subset of the values of the cached memory - if a value is present in the cache it can be fetched from there, while another pipeline stage can use the main memory or another cache of it. Nonetheless the pipeline has to be stalled if a cache miss occurs. Of course the main purpose of a cache is to overcome high memory latencies and not to resolve pipeline hazards.

### 1.2.5 Branch Prediction

The fetch unit has to know whether a conditional branch was taken and to which address control flow was transferred to. Within a pipeline this information may not be available in time: Instructions following the branch may be in the pipeline although the branch is taken. Depending on the instruction set architecture these instructions are either executed as so-called delay slots or have to be replaced by nop instructions (pipeline flush). For example the MIPS R4000 branch instruction consumes 3 cycles with one of them used as a delay slot. Branch predictors are state machines that “guess” the outcome of a branch to fetch instructions likely to be in the pipeline after a branch and thus try to avoid CPU cycles consumed by wrongly fetched instructions. A well designed fetch unit will correctly predict jumps at a high rate.
1.2.6 Advanced Pipelining

A microarchitecture may contain more than one pipeline (e.g. in case of VLIW architectures) and a pipeline may split up to allow parallel execution and then reunion again for writing results.

The assignment of instructions to different successor stages is called issue. An exemplary implementation of Tomasulo’s algorithm and extensions to it (solving problems involved in instruction scheduling (e.g. imprecise interrupts) are presented in the illuminating article [Sohi, 1990]. VLIW architectures have one fetch unit and multiple decode units but issuing may be fixed - therefore the multiple pipelines are called slots.

Figure 1.1: An advanced MIPS pipeline

Figure 1.1 illustrates an advanced MIPS pipeline presented in [Hennessy and Patterson, 2007]. It should be noticed that subpipelines may have different length and therefore instructions may reach the stage that merges the pipelines out of order - simultaneous writebacks and WAW hazards may occur. To avoid such hazards the issue stage must stall parts of the pipeline in scenarios detected by a stall logic. Reservation tables can be used by compilers to avoid stalls or by cycle accurate simulators to account for them. The advanced MIPS pipeline enables parallel execution of instructions, however every cycle at most one instruction can be issued. In contrast so-called superscalar architectures are able to issue multiple instructions each cycle - they reach a clock count per instruction smaller than one.

VLIW can be seen as a simplification of superscalar architectures - instruction scheduling is no longer done in hardware at runtime, but instead by the compiler at compile time. Multiple already scheduled instructions are fetched simultaneously - each filling a slot and being executed in a dedicated pipeline. The obvious advantage of this concept is its (hardware architectural) simplicity with the downside of increased instruction bandwidth and the dependency on high quality compiler designs.
Other advanced topics include dynamic scheduling resulting in out-of-order issue, execute and writeback (e.g. CDC 6600 processor with scoreboards) or register renaming (solving WAR and WAW hazards) - But these topics go beyond the scope of this work.
Chapter 2

Architecture Description Languages

Advances in microchip technology lead to integration densities that allow whole systems to be implemented on a single die. These system-on-chip (SoC) systems offer interesting opportunities like shorter interconnects between subsystems and thus higher bandwidths or composition of many subsystems that are optimised for the specific application. Designs can profit from such application specific optimisations (e.g. in terms of performance or energy efficiency) but they have the drawback of longer design and implementation phases (compared to commercial off-the-shelf products) resulting in higher non-recurring engineering (NRE) cost.

Especially application specific processors are challenging as design decisions not only affect the hardware itself but also tools that are needed within the implementation process of the application software like compilers, assemblers or simulators. In the phase of design space exploration these tools are typically implemented and evaluated in an iterative process. Processor architecture description languages that support automatic generation of tools have the potential to dramatically shorten this process of design space exploration.

The term architecture description language covers hardware and software architectures but will be used as a synonym for hardware architecture description languages within this work.

2.1 Classification

Several processor description languages have been proposed, many of them originate in the demand for retargetable compilation infrastructures (see [Qin and Malik, 2002]). A first categorisation can be made in respect of the level of abstraction used. The higher the abstraction of a language the more difficult the task to cover a wide range of processors - very specific or uncommon features may not be covered by the semantics - the lower the
level of abstraction the more details must be explicitly stated, resulting in longer, more error-prone and harder to read and understand descriptions.

Hardware description languages such as VHDL or Verilog allow descriptions to be very low level - for example they allow for strongly asynchronous designs but lack sufficient abstraction to explore architectures at the system level [Mishra, 2005]. Modelling languages like UML on the other hand put focus on the whole system, describing subsystems and interfaces but leave open how components of subsystems function.

A quite common level of abstraction for architecture descriptions is that of register transfer level or RTL. It is characterised by using the clock cycle as the granularity of time, which is the time between two consecutive clock events on the physical hardware. Furthermore dataflow is modelled to happen after each clock cycle (at a clock event) by moving data from the input side of a register to its output side (infinitely fast). Data manipulation operations happen during one (or more) clock cycles. For synchronous systems this representation is very intuitive and has a long tradition in digital design.

Besides different levels of abstraction a distinction can be drawn between structural and behavioural descriptions, where in fact architecture description languages will contain both with putting focus on one of them. The semantics of a “structural” language together with a given description (e.g. a netlist) imply possible behaviours of the system. A “behavioural” description together with language semantics imply possible (possibly a lot of) structures that realise that behaviour. Even classical programming languages could be used to model hardware solely behavioural, but as they support a very high level of generality every single detail would have to be hand-coded.

2.2 Related Work

2.2.1 MIMOLA

MIMOLA is a language developed in the late 70ies at the University of Kiel. With MIMOLA hardware descriptions at RT-level can be generated out of descriptions of typical applications (in a Pascal-like form) together with execution frequencies of paths through these applications and replacement rules for application elements [Mishra and Dutt, 2004]. Besides MIMOLA allows predefined resource specifications that can partly or fully describe a machine structure (through modules and their interconnects at a RT-level). Missing structures are generated by MIMOLA as a result of transforming different MIMOLA language inputs - MIMOLA is therefore categorised as a structural description language [Mishra, 2005].


2.2.2 ISDL

With ISDL (Instruction Set Description Language) an architecture is described on a behavioural level. It is designed to cover a broad range of instruction set types like VLIWs or DSP extensions and serves as a starting point for generating a compiler and an assembler [Hadjiyiannis et al., 1997]. Generality is reached through the use of C syntax wrapped into different language constructs that form several sections:

- Instruction Word Format: Names are provided for bitfields (i.e. consecutive bits).
- Global Definitions: Definitions in this section support the generation of Lex and Yacc [Levine et al., 1992]. So-called split functions allow long bit sequences (like immediate values) to be split up into multiple bitfields.
- Storage Resources: Register files, single registers and memories can be defined.
- Assembly Syntax: Parallel operations for each single instruction are defined.
- Constraints: Reduces the number of possible combinations of operations to those allowed by the architecture.
- Optimisations: This section is optional and intended to give compiler hints resulting in faster code.

Later achievements [Hadjiyiannis et al., 1999] include an instruction set simulator and present algorithms for extraction of structural information that can be used by a hardware synthesis tool. The generation of the simulator is not described in detail.

2.2.3 Expression

Expression is a description language that consists of behavioural and structural elements. Its main goal is the support of design space exploration and software toolkit generation. An interesting feature of Expression is its ability for rapid design space exploration that allows the designer to make use of reduced accuracy of descriptions resulting in coarse evaluation of alternative design candidates. With the full description it is possible to generate an optimising compiler and a cycle accurate simulator. Scheduling information for the compiler that is caused by conflicts in the pipeline is not explicitly specified in the description but computed out of the structural description. Different memory systems (including hierarchies and buses) can be specified in expression and are respected by the generated optimising compiler. The language itself consists of different sections that specify operations, instructions, components, data transfer paths and memory systems. The use of expression for automated generation of VHDL hardware descriptions is presented in [Mishra et al., 2004].
2.2.4 LISA

LISA is an architecture description language initially designed to enable retargeting of fast compiled simulators [Pees et al., 1999]. A LISA description allows to specify pipelined architectures on an operational level enabling the generated simulator to be cycle and even phase accurate. The behavioural description of one instruction is built up out of atomic operations (e.g. arithmetic or logical). Precise timing information on the other hand result from resource consumption expressed by atomic operations organised in extended Gantt charts. A restriction of LISA is the assumption that bubbles are inserted in the pipeline on resource conflicts, thus disabling simulation of out-of-order architectures. An extension to LISA is described in [Braum et al., 2004]. The generation of compilers was enabled by adding special semantics code to the description.

2.2.5 nML

nML is a mixed description language that embodies structural an behavioural descriptions. Storages (memories) are the central elements being both used as a skeleton for structural interconnects and as the representation of the architectural state [Fauth et al., 1995]. The programmers view of the architecture (including assembler mnemonics and binary encoding) is used as the level of abstraction leaving details of the micro architectures implementation undefined. Descriptions in nML are however rich enough to be used as models for automatically generated simulators [Rajesh and Moona, 1999].

2.2.6 MADL

The Mescal Architecture Description Language (MADL) described by [Qin et al., 2004] has been used to generate cycle accurate simulators, instruction schedulers and register allocators. The layered approach partitions toolset specific constructs and architecture specific functionality. A core ability of MADL is the possibility to define instructions via finite state machines. Structural and data resources are modelled as tokens passed to these state machines. Functional units can act as token managers assigning hardware resources to instructions. This describes parallelism in a natural way and covers more complex architectures like modern superscalar processors with out-of-order execution. MADL was used to generate cycle accurate simulators for different architectures.

2.2.7 PD-XML

Seng et al. propose a language with XML syntax [Seng et al., 2002]. A description consists of three types of elements: storage (e.g. register files), an instruction set description
and a resource (microarchitecture) description. The microarchitecture is described on high or low level, where one high level description can be mapped to different low level implementations - supporting PD-XMLs claim for high flexibility.

2.2.8 ArchC

ArchC [Azevedo et al., 2005] is a description language based on SystemC with a higher abstraction level intended for processor description. The two parts of the description define the architecture resources on one hand and the instruction set on the other. The possibility to verify detailed descriptions with coarse behavioural models is a useful feature of ArchC. A full blown simulator generator is able to generate cycle accurate simulators that optionally use compilation techniques. An ABI section in the description defines the application binary interface used by the simulators for handling of system calls which are used to simulate an operation system environment [Bartholomeu et al., 2003]. All tools and documentations are freely available as ArchC is developed as an open source project.

2.2.9 Generic Netlists

Architecture Description Languages can be used to generate hardware descriptions as presented in [Gorjiara et al., 2006]. In this paper it is demonstrated how an application (in form of C Code) together with an XML based architecture description (generic netlist representation - GNR) can be translated to a synthesisable (and simulateable) hardware description. The architecture description only outlines the machine, while the instructions (in form of so-called control words) are generated and stored in the synthesised memories. The presented form of hardware description is targeted to very application specific processors (“tight” hardware/software co-design) but not suited to develop general purpose cores.

2.2.10 Portable Compiled Instruction-set Simulators

D’Errico and Qin present a retargetable simulator framework that uses a behavioural description to generate interpreting and translating (both static and dynamic) simulators [D’Errico and Qin, 2006]. They highlight the advantages of retargetable and portable simulators. For their translating simulators, target instructions are converted to C++ code which is compiled with GCC to one of many supported host architectures. The dynamic translating simulator uses GCC to compile dynamically loaded libraries which are loaded at runtime. To reduce compiletime only frequently executed parts of the target application are compiled - the rest is interpreted. They include measurements highlighting speed gains obtained by using run-time translation (being approximately 4 times faster).
2.3 xADL

The xADL is a structural description language developed at the TU Vienna, Institute of Computer Languages. It uses extensible markup language (XML) syntax to describe embedded pipelined architectures. They are described by their structure in form of functional units, registers and memories connected in a network. Functional units can have user defined functionality described by atomic operations that are precisely defined in xADL. Only simple memory hierarchies (without a memory mapping unit) are supported in xADL (at least at the moment) which limits its application area to processors with simple memory subsystems often found in embedded systems. It is however possible to describe different types of caches that are characterised by their features (rather than by their explicit functionality).

The xADL language is distinguished by the following paradigm: A single compact description serves as the basis for a wide variety of analyses and generators (xADL modules). Language constructs that directly support generators by giving explicit hints are avoided. Therefore many different modules can be supported without redundancies that could make architecture changes (needed in design space exploration) cumbersome or even contradicting (which could lead to inconsistent results of different modules). Examples for this paradigm are lacking explicit listings of the instruction set or reservation tables.

The compact description shifts complexity to the analyses and generator modules - a module is responsible for extracting information out of a non-specific abstract description. As mentioned above this complexity is justified by descriptions being compact and consistent.

Another interesting aspect of xADL is its support for VLIW architectures - units can simply be duplicated by defining a repeat-count.

Besides pure hardware xADL describes the interface to the software system within a special ABI section. It defines how values are passed to functions, how registers are saved on calling functions and which registers are reserved for special purposes (like the program counter and the stack pointer). This information is especially useful for compiler related generator modules but can also be important for other modules.

Currently two different architectures are described in xADL: A MIPS R2000 derivate and the CHILI architecture (a novel 4-way VLIW architecture developed by ON DEMAND Microelectronics). While the MIPS description evolved together with the development of xADL and its modules the description of the CHILI architecture could be easily implemented afterwards without any major changes to xADL. Modules that are available by now include a generator for compiler related tables and rules, an instruction set extractor, a visualisation module (producing a network graph of the architectures structure) and a simulator generator. This simulator generator, its output and all involved technologies are the topic of this work and are described in the main chapters.
2.3.1 Syntax

As already stated, XML is the base of the xADL syntax. An xADL architecture description must contain the <adl:Architecture> top level construct, an ABI tag, at least one configuration and the netlist of units and datapaths building the architectures structure.

ABI

The ABI tag specifies the binary interface used by the software system of the architecture. All registers are categorised in classes and their use for argument passing and return values is specified. Special registers like the stack pointer are defined and optionally a name is assigned to each register in the registerfiles - for example these names can be used by modules for debugging purposes or automated generation of documentation. Memories can be classified to be data and/or program memories. It is possible to define more (named) ABI sections inside one architecture description that can be switched between by executing the xADL toolkit “adlgen” with different parameters.

Configuration

It is possible to define multiple configuration tags each of which specifies values for symbolic parameters. These parameters can be used in the structured netlist description as bitwidth parameters (e.g. for registers or buses) repeat counts of units or register counts inside a registerfile. Therefore it is possible to explore different versions of an architecture with differing parameters in just one description (e.g. 32-Bit versus 64-Bit architecture). Additionally it is possible to define components inside configuration tags to enable different versions of components in a description.

Types

Before the structured netlist can be defined in xADL the types of all used components have to be declared. These components can be immediates, memories, caches, registers or functional units. The corresponding language constructs and its parameters that declare types are described in the following list.

- <MemoryType>

  Like any other type declaration memory types must be given a name. The size of the memory is specified in bytes, its latency parameters is coarsely stated in cycles by the min-delay and max-delay properties. A number of input and output ports define the memory interface. An additional address (input) port does not need to
be defined - it is added implicitly for each input and output port. The bitwidth of ports and the number of bits used for addressing can be specified together with optional baseaddress and alignment attributes.

- **<CacheType>**
  Caches are roughly declared by their latencies (similar to memories) and a type attribute. For example such a type can be "set-associative". The exact semantics of these types is not defined in detail yet.

- **<RegisterType>**
  Registers have these main attributes: name, bitwidth and repeatcount. For this type the repeat-count is not only useful in the case of VLIW architectures. It is used to define register files. Ports define the interface of register files: they can be assigned a read-only, write-only or read/write attribute. The bitwidth attribute of ports defines how many bits of a register are accessed while the offset attribute specifies which bits are accessed. Note that the ports bitwidth may be different from the registers bitwidth and thus allows definition of subregisters (as found on x86 for example). If offset and width are omitted the full register is accessed by default. On many architectures constant registers are found. A constant tag with its index and value attributes declares a register to be constant and what value it holds.

- **<ImmediateType>**
  Immediates origin from instruction words. Although they are therefore not part of the hardware, they can be seen as data sources (similar to registers) and thus are connected to the network of units like any other read-only data component. The only attributes of immediate types are name and bitwidth.

- **<UnitType>**
  Unit types are the most comprehensive types. Like others they have a name and contain ports defining their interface. Units are the components that implement a functionality defined by their operations that themselves consist of micro operations (built-in calls). Furthermore units can encapsulate networks of other components like registers, immediates or even (sub) units to support hierarchical design - such an encapsulating unit may not define operations as they are only permitted on the lowest level of the hierarchy.

  Operations can process information delivered to input ports of units and specify how outputs are computed. It is possible to define several operations in a single unit. This means that these operations are alternative computations that can be performed by the unit. An example for this are different arithmetic operations performed in an ALU unit type.

  Besides inputs, outputs and constants, temporaries serve as additional parameters for built-in calls. They are declared inside the unit type and have name and bitwidth.
attributes, constants carry a value. The following section describes operations in detail.

Operations

As mentioned before operations are declared within unit types. Apart from a name operations have two specific attributes - syntax and binary which are described in Section 2.3.2. The building blocks of operations, named calls, are collected within body tags. Several predefined built-in calls are available to model a units behaviour. Additionally it is possible to implement user defined calls but their exact functionality and semantics are left open at the moment.

To define (complicated) operations several built-in calls can be combined - it is however not possible to nest calls. Instead temporaries are needed to reuse results as built-in call arguments. There is one distinguished built-in call to model conditional functionality: cmove. The functionality of the conditional move call is as follows: an argument is evaluated to true or false (like in C its value is compared to zero) and depending on that one of the two additional arguments is stored as the result of the call.

Instantiation of Types

This part of the xADL description forms up the concrete network out of instantiated components and connections. Typically it begins with instantiating the data sources (sinks) already mentioned: registers, immediates, memories and caches. These instantiations have in common to need a name tag and one to reference the instantiated type. For caches the memory instance that is cached needs to be stated. Registers have a category property that defines special purposes (e.g. integer, base, etc.).

However the major part of the instantiations is formed up by units. They are the only components allowed to use the connect tag that specifies the datapaths between components and thus the architectures structure. The syntax of these connect tags allows to specify separate input and output connections for the interface of the corresponding unit. Both of them use the select attribute to reference the other end of a connection (inputports for output connections and vice versa). When two functional units are connected to each other, it does not matter whether this is described by an input connect in one unit or an output connect in the other.

An important property of connections is declared by the stageboundary attribute: It gives connections the semantics of transferring data only on clock events (modelling of hardware register transfer). Such connections are used to split components that belong to different pipeline stages.

Hazards have a similar syntax to connections, but quite different semantics. They are used
to model (virtual) connections between units, that are used to resolve pipeline conflicts. The forward hazard is a connection that forwards a result from an output port to an input port and thus resolves data hazards. The stall hazard, if connecting an input and an output port has a similar purpose, but instead of forwarding a value it has the semantics to stall the pipeline until the result is available at the input port (e.g. until it is written back to the register file). If the stall hazard connects two output ports its semantics is to resolve WAW conflicts. The ignore hazard is a dummy signalling a hazard not being resolved in hardware (e.g. this information can be used by a compiler generating module to resolve conflicts in software).

2.3.2 Instruction Set

As already mentioned the instruction set is not defined explicitly. The instruction set is extracted by following all possible paths in the network using a breath first search. Each path represents a potential instruction of the architecture. But typically not every path corresponds to an instruction, thus xADL uses a predicate/condition concept to exclude paths. Predicates and conditions can be defined along data paths inside units, unit types and operations. The condition tags evaluate predicate tags defined earlier on the path and omit instructions that do not satisfy those conditions.

An example for such condition is to use a restricted set of operations in an ALU for address calculation: A predicate addressable could be set for addition and subtraction - an appropriate condition in a memory access unit could forbid all other ALU operations.

Syntax and Binary Representation

When describing an existing architecture the syntax or at least the binary encoding of instructions have to be specified to make fully automatic toolkit generation possible. In xADL the syntax of instructions is described constructively along the instruction path. In a syntax section one or more syntax templates specify different types of instructions. They may consist of strings and several tokens. Most components in the network (even connections) may use syntax tags that specify concrete values for tokens. Those values can be strings, values of immediates or indices in a register file.

The binary encoding of instructions is defined similarly with templates and tokens. Additionally bitmasks select characteristic bits of an instruction word. The values for these masks are assigned just like tokens along the instruction path.

This form of syntax and binary description is especially suited for orthogonal instruction sets, often found in embedded processors, because only few different syntax templates are needed.
Chapter 3

Simulator Generator

This chapter focuses on the automated generation of simulators for an architecture described in xADL. The architecture to be simulated is called “source”, while the architecture that runs the generated simulator is called “host”. There are many purposes for which simulators can be useful: (old) applications that are written for the source architecture can be used on (modern) hosts even when the sourcecode of these applications is not available or difficult to port; debugging is possible at a very low level and performance bottlenecks of applications can be detected at the microarchitectural layer where cache misses or pipeline stalls are relevant for execution speed.

When a simulator is generated from a high level description language it is easy to change important parameters of the architecture, like the number of registers or the number of functional units. To evaluate the properties of the resulting architecture, benchmarks compiled for this new architecture are needed. An automated generation of compilers from the same description language makes such benchmarks possible and enables to evaluate completely new architectures even in their (early) design phase. The xADL has the potential to provide these features and parts of them are already finished.

The simulator generator presented in this work supports the generation of both an interpreter and a dynamic binary translator. For code generation of the translator the LLVM framework is used; it provides a generic interface to produce code for a variety of host architectures and is frequently improved by its active developer community.

The core of the simulator generator is implemented in form of a module for the xADL framework. This framework (among other things) is able to extract the instruction set of a specified architecture. The simulator builds on this instruction set that specifies in detail when (in which pipeline stage) and which operations are to be performed by an instruction. This information is then transformed to an executable form and enriched by the operations needed for the simulation of the whole architecture. Besides the executable description of the instruction set, other important architecture specific parts, like the
instruction decoder or the virtual register files, are generated to complete the simulator.

3.1 Simulator Framework

The source code of the simulator is not completely generated out of the architecture description language. This chapter explains the hand-written framework that contains architecture independent parts of the simulator. Important parameters for this framework like the number of pipeline stages and slots or the sizes of memories are set by pre-processor macros in the generated code.

3.1.1 Simulation Model

The core of a classical interpreting simulator consists of a loop iterating through simulated instructions of the source architecture. An instruction is loaded from the simulated program memory, decoded from its binary representation and then executed: The simulated registers and memories are altered according to the semantics of that instruction.

It is possible to implement simulation at different levels of abstraction: Instruction set simulators provide a semantic model of the instruction set but ignore the source architectures internal structure. While such simulators may perfectly run binary applications they are inherently inaccurate when exact timings (cycle count) of the simulation are needed. Especially for architectures that have memory hierarchies, multiple pipelines or other features that dynamically influence the execution speed of an application, at least some parts of the architecture’s structure have to be taken into account to make cycle accurate simulations possible.

At a very low level of abstraction (e.g. transistor level simulation) internal timings even for physical phenomena like signal propagation can be taken into account.

Of course the simulation speed depends on the level of abstraction: Generally a lower abstraction level leads to lower simulation speed.

We use pipeline simulation that operates on a moderate level of abstraction and features an explicit model of the architectures pipeline structure. In that case one iteration of the simulators main loop does not simulate the semantics of a whole instruction but that of a clock cycle (shifting instructions through the pipeline).
3.1.2 Architectural State

The state of the simulated architecture consists of all its registers and memories. Besides registers that are explicitly defined in the xADL (see Section 2.3) the architectural state includes additional registers like pipeline-registers (registers between two adjacent pipeline stages) or forward-registers. While some of those registers may not even exist on the physical processors they are needed for their simulation. This is mostly because pipeline stages are executed simultaneously on a physical processor, but have to be simulated sequentially in the simulator. One aspect of this discrepancy is that physical registers might be used as input and output in the same clock cycle by different units. In many cases such simulation conflicts can be resolved by an appropriately ordered sequential execution of the corresponding pipeline stages. When no such ordering is possible additional (shadow) registers are required for the simulation.

Simulated instructions operate on the architectural state, by moving values between registers and/or modifying them according to their semantics. Each instruction can be decomposed to smaller operations and all such operations, that are performed in one pipeline stage form the so-called pipestage-function of an instruction for this pipeline stage. For each instruction and pipeline stage a separate pipestage-function is generated in form of a C-function that performs the appropriate operations on the global architectural state.

3.1.3 Pipeline Organisation

Beside the architectural state, the pipeline buffer is another important data structure used to represent the pipeline’s logical state that is basically organised in form of a ringbuffer. Each entry in this buffer corresponds to an instruction currently in the pipeline. For VLIW architectures it is two-dimensional - one dimension for pipeline stages and the other one for slots.

Each entry in the pipeline buffer contains the following items:

- The type of the instruction (e.g. add immediate)
  This is technically a pointer to a generated structure that provides all stage-functions for the instruction, an instruction id and information about the instructions ability to change the programflow.

- The operands of the instruction
  These contain all register indices and immediate values of the instruction and are provided as arguments to the stage-function of instructions each time they are called.

- The program counter
It specifies the position of the instruction in the program memory and is mainly used by the dynamic binary translator to enable caching of basic blocks.

### 3.1.4 Pipeline Simulation

The pipeline (ring) buffer is the essential part of the simulator core loop. In each iteration a new instruction is inserted into the buffer by overwriting the oldest entry. Each instruction enters the pipeline at stage 0 and propagates to the end of the pipeline while its pipestage-functions are executed modifying the architectural state. The instructions position within the pipeline buffer determines which of its stage-functions is called.

**An Example:** The MIPS R2000 architecture is specified in xADL and provides an instruction ”add immediate” with the internal id 69. The stage-functions generated for this instruction are therefore called ”instr\_69\_stage0”, ”instr\_69\_stage1”, ”instr\_69\_stage2” and so on. When an ”add immediate” instruction completely shifts through the pipeline (which requires 5 cycles) all its stage-functions are called one after the other - each with the operands (arguments to the C-function) specifying the immediate value to add and the source and destination register indices belonging to that instance of the ”add immediate” instruction.

To simulate a full clock cycle all of the instructions currently in the pipeline have to be processed one after the other beginning at the back end of the pipeline. This ordering is chosen to overcome the need for a lot of additional registers (see Section 3.1.2).

However not all of the simulation is done in just one pass. To model special structures, like forward registers or registers that are concurrently written in different stages with different priorities, shadow registers are needed. This is because the execution order, that would be needed to avoid those shadow registers, is conflicting with the back-to-front execution order of the pipeline buffer. The contents of shadow registers have to be copied back to their corresponding origin registers (writeback) after the pipestage-functions for a full cycle are executed.

Therefore the generator has the ability to generate code for a second pass - each stage-function has its so-called epilogue that can be used for the writeback and gets executed in this pass.

In each cycle the decoder inserts a new instruction into the pipeline buffer. The next instruction to decode is determined by the source program counter (PC) which is normally used to index the simulated program memory. Jump instructions (and others that modify the programflow) change the PC in their stage-functions (or epilogues); if no such explicit modification takes place in a simulated cycle the simulator framework increments the PC by the size of the last decoded instruction - and thus the subsequent instruction gets decoded in the following cycle.
3.2 System Calls

Architectures found in embedded systems are the main target of the xADL description language. Such environments, especially for performance critical applications, sometimes lack an operating system layer between hardware and application software - interactions with peripheral hardware must be implemented directly using memory mapped I/O or special instructions then.

Nonetheless to enable a generic interaction between the simulated program and the host system, we use special system calls (syscalls) in our simulator. We provide a subset of newlib (a C library intended for use on embedded systems) syscalls. By definition the source applications initiate the execution of such syscalls by calls to special addresses. Before an instruction is decoded the current PC is compared with these addresses. In case of a match, special instructions are inserted to the pipeline buffer that contain the execution of the corresponding syscall and an instruction sequence that provides the return to the calling site of the syscall. Parameters to (and return values from) the syscalls are passed through the registers specified in the ABI section of the xADL description (see Section 2.3.1).

We use compilers configured with newlib to compile executables for our two example architectures (MIPS R2000 and CHILI) while the simulator itself is compiled with the “GNU C Library” (glibc). The two C libraries differ in naming and binary encoding of flags and parameter ordering and we have therefore implemented wrapper functions for the most important syscalls, that translate source (newlib) syscalls to host (libc) syscalls.

3.3 Binary File Loading

The simulator is capable of directly loading binary ELF (Executable and Linking Format) files, that can also be used to run on real targets. It uses the Binary File Descriptor library (libbfd) [Pesch and Osier, 1993] for this purpose and therefore an appropriate version of libbfd must be available on the host system to simulate a certain architecture. To support the loading (and execution) of binary files of architectures that are not available or still in the development phase it is (theoretically) possible to generate suitable libbfds along with the corresponding compilers out of the xADL description.

A (simplified) ELF file contains a header and a list of sections that contain binary data. To execute such files, the contents of these sections are first copied to the simulated program and data memories in the architectural state and then the simulated program counter is set to the starting address specified in the header section. Currently the simulator framework supports two separated (continuously addressed) memories for data and program sections. One of these memories is chosen depending on the type of the section to load:
sections that contain executable code go to program memory while the remaining ones are loaded to the data memory (Harvard architecture). It is however possible to define one memory for both program and data sections, which allows the simulation of Von Neumann architectures (e.g. the MIPS R2000). For more complicated memory systems, with multiple data memories and/or fragmented address space some improvements to the current implementation of the framework are needed.

### 3.4 Internal Description

To generate a simulator for an architecture out of a structural description it has to be transformed to a (more or less) sequentially executable form. For our work this means that all implicit and explicit semantics of the description are mapped to a sequence of executable instructions and control structures that explicitly describe how to implement the desired behaviour. For example the structure of a forward link is represented through an instruction/control sequence at each end of the link: A store instruction to a forward register at the back end and a conditional read instruction of the same forward register at the front end.

For the interpreting version of our simulator these instruction sequences are generated as C-code, that directly operates on the architectural state which is implemented in form of global C variables. In a first approach the generation of C-code out of the description was made more or less directly in one pass, printing one or more C-statements for each operation along the extracted instruction path (see Section 2.3.2 and [Brandner et al., 2007] for instruction path extraction). The simulator generated that way worked but needed a lot of special cases in the generator to cope with all the semantics possible in the xADL description (and even did not cover all of them). But the main disadvantage of this first version was its lack of extendability.

Our goal was to extend the purely interpreting simulator to a (partly) translating one. The idea was to generate C-code that instead of directly modifying the architectural state, would itself generate (machine) code that performs these modifications. To achieve this goal we use the LLVM framework, which offers a C++ interface to generate the needed machine code [Lattner and Adve, 2004].

So the simulator generator needs to generate two forms of executable sequences, which have completely different appearance but very similar structural composition. To accomplish this, without duplicating big parts of the generator code, we introduced an internal description of (pseudo) executable sequences.

In a first phase the generator translates the xADL description to this internal description which includes explicit operations to cover all architectural properties to be simulated. The actual generation of the two simulator parts (interpreter and binary translator) is
carried out in the second phase where each element of the internal description is generically translated to the desired form of code. Because basic elements of the internal description have simple and well-defined semantics this translation can be performed without any special cases and independent from other elements.

Besides a rough overview, Figure 3.1 illustrates some details of the internal description: Sequences and memory-elements which are described in the following sections.
3.4.1 Memory-Elements

The internal description can be seen as virtual programming language: calls (that are closely related to those in the xADL description) form instructions of this language. Variables for this language are called memory-elements or mem_elements.

Each mem_element has a type and some parameters that specify its properties. One important property is the bitwidth of the memory-elements which may be an arbitrary positive integer. Memory-elements may be constant and have different visibility scopes: Basically we distinguish between global and local mem_elements but for the latter ones two versions exist.

Memory-elements with global scope have a unique identifier and are visible within all pipestage-functions of all instructions. Global mem_elements are eventually translated to global C variables. They are used to represent the architectural state (mainly registers and memories) of a described architecture.

The first kind of local scope is called temporary, is local for a single pipestage-function and is translated to a local variable for the interpreter. The second kind is operand scope which is used for instruction operands that can be immediate values or register indices and are visible (and have unique identifier) within an instruction.

Besides "normal" memory-elements that store a single value with the specified bitwidth there are two types of array memory-elements: registers and memories. For both types an additional parameter specifies the size of the array in registers or bytes (for memories). Instead of using array elements directly (as arguments or results) in the call sequences, special array access mem_elements are needed. These can be used like any other memory-element and encapsulate the array and an index, itself a memory-element, that specifies the position to be used inside the array.

3.4.2 Call Sequence

Call sequences are a simple sequential succession of calls (basic operations) and condition elements. Calls have a number of mem_elements as arguments and results and perform one of a few predefined logical and arithmetic operations on them. The semantics of these operations is identical to those of the xADL built-in calls, because most of them are translated one-to-one. Other than in the xADL description, there is no cmove (conditional move) call in the sequences of the internal description, because control flow is modelled explicitly in form of condition elements.

The structure of condition elements is very simple: they consist of a condition and two call sequences (if_sequence and else_sequence). At runtime one of these sequences is executed depending on the value of the condition. With these simple control flow elements it is
not possible to represent an arbitrary control flow graph (CFG) but only simple or nested if-then-else structures. For the translation of operations containing cmove calls and other simple conditions this strictly hierarchical representation is sufficient and enables easy translation to C-syntax.

### 3.4.3 Condition Tree

Conditions in condition elements always compare memory-elements (that must have matching bitwidth parameters) pairwise with each other. Possible relational operators that may be used for their comparison are:

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Operation</th>
<th>Semantics</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>equal ((A = B))</td>
<td>lt</td>
<td>less than ((A &lt; B))</td>
</tr>
<tr>
<td>neq</td>
<td>not equal ((A \neq B))</td>
<td>ge</td>
<td>greater or equal ((A \geq B))</td>
</tr>
<tr>
<td>gt</td>
<td>greater than (\text{unsigned } (A &gt; B))</td>
<td>le</td>
<td>less or equal (\text{unsigned } (A \leq B))</td>
</tr>
</tbody>
</table>

**Table 3.1: Possible Condition Operators**

It is possible to build more complicated conditions by combining simple comparisons with logical AND and OR operations. Together with constant memory-elements the resulting condition tree has the expressiveness to represent all needed (unsigned) conditions.

### 3.4.4 Building up Internal Description

As stated before the call sequences are built according to the operations along the instruction path of the xADL description. Many of these operations (more precisely their calls) are directly translated in sequential order. For operations that include cmove calls or additional architectural semantics that is not directly represented in the instruction path (e.g. register write priorities) a more complicated translation is needed that might involve condition elements and non sequential description construction. While the description of a specific instruction is generated an insertion pointer defines the position inside the call sequence, where the next call is to be inserted. For direct sequential translations this pointer is just directed to the next position. For conditional instructions (e.g. "branch not equal" on MIPS R2000) parts of the operations have to generate their calls inside the if_sequence of a condition element - this is achieved (transarently) by setting the insertion pointer to that sequence.

In more complicated situations, where mem_elements are affected by multiple cmove calls, condition elements can be transparently combined with logical AND or OR and form new condition elements.
Chapter 4

Simulator Generator Implementation

The previous chapter introduced the simulator generator and its internal description - it sketched the basic concepts. The implementation of those concepts not only accounts for great amounts of working time but also reveals interesting aspects of generators and especially (pipeline) simulators. This justifies the generator implementation to be discussed in the main part of this work.

The simulator generator is implemented as a module of the adlgen tool. The main function of the module consists of two calls: build_description() which builds up the internal description mentioned above and emit_code() that transforms the description to C++ code that is finally written to files (and forms the architecture dependant basis of the simulator).

The top-level build_description() function works on instruction level in the first place by calling the build_description() function of each instruction object (the instruction objects with their built-in call lists are prepared by the adlgen tool). The generator module follows the datapath used by an instruction (represented by the built-in call list) and extracts information on-the-fly.

At the back-end that translates the internal description to C++ code, polymorphism is used to execute an appropriate emit_code() method for every object of the internal description. Depending on what source code file and what sort of code is intended to be produced, emit_code() is called with different parameters (the file pointer to the output file and a code type parameter).

The implementation of the generator front-end that builds up the internal description is discussed in Section 4.2 that sketches out basic concepts of the generator and Section 4.3 that explains the translation of pipeline hazards. But first of all the structure and contained datatypes of adlgens so-called web are to be examined.
4.1 The Web

In Chapter 2.3 the xADL language with its components and connections has been described. The adlgen tool takes an xADL description as input and translates its components and connections to an object-oriented internal representation that is called the web.

Only a rough overview of how the web is built up will be given as for the generator the resulting structure is of greater interest. The following enumeration lists the major methods used in adlgen to build up the web. The MI parameter passes the model_information object - it serves as a root object to make the web accessible.

1. **create_xml(MI)**: Uses libxml [Veillard, 1999] to read in the xADL description and validate it against the xADL schema. Applies the selected configuration to supply parameters with fixed values.

2. **web_create(MI)**: Iterates through all XML elements of the description and creates objects by calling `create()` of the appropriate classes (e.g. `register_type_t::create`).

3. **abi_create(MI)**: Extracts register purposes, register naming, data and program memories (ABI elements as explained in Section 2.3.1).

4. **instruction_create(MI)**: As the architectures instruction set is not explicitly stated in an xADL description this method is used to extract it. A breath first search finds all data paths through the web, each getting an instruction (if not prohibited by a predicate/condition relation). A delay and latency analysis is performed - its results (a lower and upper bound for the execution time of each instruction) are used by the compiler generator module.

4.1.1 Components

The components contained in the web can be broken down into various categories. Components that can be connected with each other are represented by objects derived from the `connectable_t` class within the web. These so-called connectables play an important role for the simulator generator.

The classes `constant_t`, `immediate_t` and `temporary_t` are derived from `connectable_t`, their instances represent simple data storages in the web. Their common property is a bit width, constants additionally have a value property. While constants and immediates serve as pure data sources, a temporary can act as both - data source and sink. A typical use of temporaries is for passing intermediate data between subsequent built-in calls.
Both register indices and immediates are instruction operands but register indices are in contrast to immediates not directly represented by web components (register indices are only present in syntax and binary templates - see Section 2.3.2 and 5.2.2).

Another class derived from `connectable_t` with further subclasses is `port_t`: Memory data and address ports, register ports and unit input respectively output ports (to name the most important ones) are represented by classes derived from `port_t`.

Although not derived from the `connectable_t` class a call can be seen as a subcomponent having input ports (arguments) and output ports (results) that are either connected with a components input or output ports or with another call. Calls are atomic subcomponents that give the web functional semantics.

### 4.1.2 Built-in Calls

As already mentioned, calls are used to build up operations that model the behaviour of units. They can either be built-in calls or calls to external functions (which are not used at the moment). Calls within the web are objects of class `call_t`. Besides an ID a call contains two lists: One for arguments and one for results. The lists contain pointers to connectables, linking the call into the web. It should be noticed, that calls may produce multiple results (e.g. for overflow and carry - see Table 4.1).

### 4.1.3 Instructions

For the simulator generator the output of adlgens `instruction_create(MI)` is of special interest: It generates a list of instructions, each containing a list of built-in calls. This list of built-in calls models the behaviour of a whole instruction. As the instruction is extracted out of a datapath through the connected components, not only calls that are explicitly stated in the xADL description are contained, but also calls that model data transfers.

An important observation is that an instruction set simulator could be directly created out of this list of calls by simply translating each built-in call to C code (the built-in calls have well defined and simple semantics). To generate a simulator operating on pipeline level, a more complex process described in the following sections was evolved.

### 4.2 Basic Principles

A basic idea behind the simulator is to model each pipeline stage within several C functions: One that implements the behaviour of the pipeline stage (stage-function) and one
CHAPTER 4. SIMULATOR GENERATOR IMPLEMENTATION

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Results</th>
<th>Name</th>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>a</td>
<td>d</td>
<td>rol</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>cmov</td>
<td>cond, a, b</td>
<td>d</td>
<td>ror</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>sext</td>
<td>a</td>
<td>d</td>
<td>shl</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>zext</td>
<td>a</td>
<td>d</td>
<td>shr</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>trunc</td>
<td>a</td>
<td>d</td>
<td>ashr</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>ceq</td>
<td>a, b</td>
<td>d</td>
<td>abs</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>cneq</td>
<td>a, b</td>
<td>d</td>
<td>add</td>
<td>a, b</td>
<td>d, o, c</td>
</tr>
<tr>
<td>clt</td>
<td>a, b</td>
<td>d</td>
<td>sub</td>
<td>a, b</td>
<td>d, o, c</td>
</tr>
<tr>
<td>cltu</td>
<td>a, b</td>
<td>d</td>
<td>div</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>cle</td>
<td>a, b</td>
<td>d</td>
<td>divu</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>cleu</td>
<td>a, b</td>
<td>d</td>
<td>rem</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>cgt</td>
<td>a, b</td>
<td>d</td>
<td>remu</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>cgtu</td>
<td>a, b</td>
<td>d</td>
<td>divrem</td>
<td>a, b</td>
<td>div, rem</td>
</tr>
<tr>
<td>cge</td>
<td>a, b</td>
<td>d</td>
<td>divremu</td>
<td>a, b</td>
<td>div, rem</td>
</tr>
<tr>
<td>cgeu</td>
<td>a, b</td>
<td>d</td>
<td>mul</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>and</td>
<td>a, b</td>
<td>d</td>
<td>mulu</td>
<td>a, b</td>
<td>d</td>
</tr>
<tr>
<td>or</td>
<td>a, b</td>
<td>d</td>
<td>mult</td>
<td>a, b</td>
<td>hi, lo</td>
</tr>
<tr>
<td>xor</td>
<td>a, b</td>
<td>d</td>
<td>multu</td>
<td>a, b</td>
<td>hi, lo</td>
</tr>
<tr>
<td>not</td>
<td>a</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Predefined built-in functions.

that gets executed in a second pipeline simulation pass (epilogue) to handle memory and register port priorities (see Section 4.3.3). Two versions of C code for pipeline stage-functions and epilogues are produced: One version directly modifies the architectural state (for the interpreter) and the other generates so-called LLVM bitcode (for the compiling simulator as described by [Rigler, 2008]).

First, for an instruction an empty pipeline stage data structure is created. This data structure mainly is a sequence in the internal representation (described in Section 3.4.2). This sequence is filled by iterating through the built-in calls and performing the following tasks for each:

1. Convert dynamic conditional moves (cmov built-in calls) to static if-else sequences (see Section 4.2.6).

2. Insert the so-called environment sequences (they mainly handle hazards and are explained in Section 4.2.3).

3. Add the built-in call itself to the sequence if not prohibited (the call may be replaced by its own environment).
4.2.1 Pipeline Stage Extraction

Each built-in call has a parent pointer that determines the operation it belongs to - the parent operation holds a pointer to its parent unit. During the instruction extraction pass of adlgen the unit has been assigned a stage number which in turn is used for a control break. This control brake ends the current pipeline stage and creates a new one for the current instruction. The details of how units are given stage numbers in the adlgen tool are not in the scope of this work.

As the list of built-in calls for one instruction contains moves that model data transfers across pipeline stage boundaries, this control break occurs just at the right place (after copying values to output ports of the previous unit and before copying them to the unit input ports of the next unit respectively pipeline stage). Listing 4.1 illustrates the call sequence of a MIPS addi (add immediate) instruction, a control break is detected after line 3.

Listing 4.1: Stageboundary in List of Built-in Calls

```plaintext
1  move DE__pc_epc_o, DE__pc_epc_i   // unit DE writes output ports
2  move DE__Rs_o, DE__Rs_i
3  sext DE__ImmWs_o, DE__ImmW_i
4  // a new stagefunction is created
5  move EX__ImmWs_i, DE__ImmWs_o   // unit EX writes input ports
6  move EX__Rs_i, DE__Rs_o
7  move EX__pc_epc_i, DE__pc_epc_o
```

4.2.2 Component and Connection Translation

Built-in calls are translated to sequence elements that are stored in the pipeline and pipeline stage descriptions of the simulator generator. It was already mentioned that built-in calls have argument and result lists of pointers to objects derived from connectable_t. As the calls get translated to sequence elements, these connectables also get converted: They become memory-elements of the internal description (see Section 3.4.1) that are finally translated to C variables by the generators backend.

This translation pays attention to the concept that all connectables in the web are part of the simulator architectural state. In fact this models behaviour of synchronous hardware implementations of microarchitectures very well: All ports connected by wires hold a well-defined value - at least at clock events - they have a state. This allows for simulation at register transfer level. A somewhat similar concept can be found in VHDL simulators: The simulation state comprises all VHDL signals. In contrast to our simulator the
state is calculated for arbitrary events, not only clock events (event-driven vs. cycle-time simulation [Ramachandran and Johnson, 2007]).

For generating the architectural state in the internal description, constants and temporaries can be directly translated to memory-elements. Ports of certain components (memories and register files) must be handled by specialised memory-elements that encapsulate how the state of the actual component can be retrieved.

**An example:** The xADL demands a memory of an architecture to have an address and a data port. Assume the adlgen tool extracted a write instruction that uses the write built-in call within a memory unit to write to the architectures memory.

The simulator generator uses the write call to create a (write) sequence element and several memory-elements. For the concrete example the memory-elements represent the architectural state that is necessary to resolve the memory access in the simulator: One memory-element represents the data port and another one represents the address port.

The generator backend will translate these two memory-elements to two (global) C variables in the simulator. Within the simulator, the architectures memory is located in an array. The value stored in the data port C variable must be written to the array at the index stored in the address port C variable. But one C variable is still missing: The pointer to the array - the base address.

This is why the generator extends the address memory-element by a pointer to an additional special memory-element that represents the base address. The C code finally produced involves all three C variables. The (simplified) generator output is:

\[
* \left( \text{uint8_t } * \right) \left( \text{Memory } + \text{DMem__ATwrite.getUVal()} \right) = \text{DMem__write.getUVal}();
\]

### 4.2.3 Call Environment

As already mentioned the internal description is built up by iterating over the instructions built-in call lists. For every call the connectables involved are translated to memory-elements. But this translation may demand for additional memory-elements and sequence elements (as the memory example above already indicates). Some other scenarios demanding for extra effort in memory element translation are: Translation of conditional moves, memory and register accesses and the translation of hazard links. Before dealing with the difficulties in those scenarios the general concept to extend the simple memory element translation is to be examined.

The chosen approach was to implement a routine for every class derived from `connectable_t`: The connectable object can be requested to create a memory element and is responsible to insert all necessary code to make this memory element valid. This necessary code may be inserted before or after the current sequence element and is called
access environment.

The methods that build the access environment are named build_access_environment(). As each may be executed for different calls, the so-called call context object is passed as a parameter. It contains all information depending on the current call: the insertion point (the current sequence and sequence element) and necessary hash tables (e.g. for active conditions) and thus allows a build_access_environment() method to build an appropriate description of the call in the given context.

The classes that build an access environment are: unit_input_port_t, unit_output_port_t, register_port_t and memory_address_port_t. Their environment building methods have in common to cover hazards explained in later sections. Therefore some details are covered in the sections dealing with hazard translation (Section 4.3). For now the following example for register_port_t should illustrate the concept.

**Constant Registers:** Register values are made constant by forbidding write access to indices specified to be constant in xADL. Therefore the environment building method of the register port creates a condition sequence element with a condition tree that checks the register index in the operand structure not being equal with any constant index. Further environment sequence elements and the register write access itself (a move sequence element) are placed into the "if" sequence and are thus only applied for indices that are not marked as constant.

### 4.2.4 Memory Element Handling

A concept of the generator that makes memory element handling more convenient is that of pools. The call context organises memory-elements in different pools, paying attention to generated C variable scopes:

- **global_pool:** It is located in the description object of the context and contains all global memory-elements. Note that unit input ports memory-elements do not need to be global as they are only used to copy in values.

- **operand_pool:** Located in the instruction description its memory-elements are used to represent accesses to the operand union.

- **pipestage_pool:** Located in the pipestage description object it contains pipeline stage-function local memory-elements and pointers to all other memory-elements (for programming convenience).

All pools are hash tables that are indexed by the (unique) memory-elements name (this name is also used for the final C variable names). The creation of memory-elements is transparent: On the first invocation of get_mem_element(string name) (method of class
4.2.5 Operand Handling

Operands of an instruction (like immediates and register indices as described in Section 1.1.3) can be used by built-in calls in any pipeline stage. Therefore the instruction operands are passed via a parameter to every stage-function generated. To unify the C function header of these stage-functions generated (irrespective of the instruction) a union encapsulating all possible operand names was used. Listing 4.2 shows a generated stage-functions header and operand handling. Of course when the pipeline is simulated every pipeline stage-function currently active in the pipeline must provide its own union (different instructions concurrently in the pipeline have different operands).

Listing 4.2: Operand Handling in Stage-Functions

```c
void instr_34_stage1(union_ops *ops)
{
  fixint <16> DE_ImmW_i1_1;
  fixint <32> DE_Rs_i1_1;
  fixint <32> DE_Rt_i1_1;
  DE_ImmW_i1_1 = ops->ops_34.ImmW; /* move */
  DE_Rs_i1_1 = R[ops->ops_34.Rs.getUVal()]; /* move */
  DE_Rt_i1_1 = R[ops->ops_34.Rt.getUVal()]; /* move */
  ...
}
```

4.2.6 Conditional Move Conversion

Multiplexers are used in hardware to direct dataflow depending on a control signal. A selectable input port gets connected with a single output port. In synchronous systems it is useful to feed back the multiplexers output (from the previous clock cycle) to overwrite old data only if a condition gets true. The xADL language allows to model such a behaviour by using the \texttt{cmove} built-in call. Depending on a single bit condition it copies one of two arguments to the result (combining \texttt{cmove} calls allows selection out of multiple inputs). Additionally the xADL language defines the \texttt{inactive} value that, if selected, does not modify the result.

This model strongly places emphasis on the data flow while classical programming models favour modelling of control flow (by conditional execution of branches and subsequent data moves). Thus the conditional dataflow is translated for the generated simulator to
if/else control structures (as illustrated in Figure 4.1). At a first view this translation seems trivial, but it should be noticed that once selected the inactive propagates through subsequent calls via read after write relations.

Within the simulator this propagation of inactives can be modelled as an additional bit for each memory element. If the bit is set, the generated sequence element does nothing but marking its result memory-elements as inactive. In the generated C Code this would result in an "if" for every translated sequence element in a read-after-write relation with an inactive, causing a significant performance overhead. Therefore a static resolution of inactives was chosen.

A \texttt{cmove} connected to an inactive is translated to a move sequence element contained in the "if" part of a conditional sequence (as described in Section 3.4.2). The condition is called \textit{active condition} for the result memory element and is stored in a hash table that can be queried for subsequent sequence elements. The condition propagates during generation time over the read after write related memory-elements. A key issue of that method is the propagation of active conditions over pipeline stage boundaries: Typically the conditions are one bit temporaries in the web resulting in local C variables (stage-function scope).

The propagation of active conditions over pipeline stage boundaries (and thus over clock cycle boundaries) is detected and the corresponding memory element (the condition) gets \textit{globalised}: It gets a global C variable. It should be noticed that changing the scope of a variable alone is not sufficient for correct behaviour. Of course reverse execution of pipeline stages as implemented in our simulator correctly handles re-usage of the global value in the next cycle, but as Figure 4.2 illustrates the global value cannot be reused after two cycles.

To solve this problem we decided to duplicate globalised memory-elements in the stage that reuses them. The new memory-elements name is suffixed with the stage number and a move sequence element is generated to copy the value from the globalised element. As an optimisation the sequence element and the new global value only get added, if they are reused in a later stage (within the generator this results in the need to handle globalised values in a separate instruction description pass). Of course this process can be repeated for value propagation over several stages.

\textbf{Figure 4.1:} \texttt{cmove} Translation
To summarise the concrete implementation, for conditional moves the following steps are performed:

1. For the current call check if its ID is `cmove`, if so perform step 2.
2. For all non-inactive arguments perform steps 3 and 4.
3. Create a condition sequence element with the following condition: Condition is true for argument 1 (respectively false for argument 2) and active condition of the arguments (found in a hash table in the generator call context) is true.
4. Insert a move in the if sequence: argument 1 (respectively argument 2) $\rightarrow$ result.
5. If the current instruction is fully translated, insert memory-elements and moves that duplicate global conditions for pipeline stage boundaries.

### 4.3 Hazard Translation

#### 4.3.1 Resolving Forwards

The xADL hazard type `ignore` is the trivial case, it does not need any translation. A typical forward hazard (see Section 2.3.1) for an arithmetical unit may be defined like in Listing 4.3. It is a virtual connection between an output port and an input port.

Figure 4.3 is intended to illustrate what information is needed to resolve the forwarding link in the call environment. To simulate a runtime forwarding decision it must be checked
whether to take the register values read in the decode unit or the forwarded value produced by the preceding instruction at the relevant unit output port. The decision to take the forwarded value demands for two conditions to be true:

1. The register index of the needed value is equal to the one of the forwarded values destination register.

2. The forwarded value is exactly one cycle old.

The two register indices involved can be found in the operand structure of the current instruction (\(ops \rightarrow ops\_X\) with X being the instruction number) and in the operand structure of the preceding instruction (\(ops \rightarrow ops\_Y\)). Note that the preceding instruction may not be in the pipeline anymore, but in practice no forward from the last unit in the pipeline is needed as it typically writes the register file.

In the concrete simulator implementation two additional "shadow" register files are used: one for the forwarded values and one for timestamps signalling validity of values. The instruction writing to an output port that is connected to a forward hazard writes the value to a forward register file and the current cycle count to a timestamp register file while an instruction reading from the input port simply checks if the timestamp for the register index is only one cycle old.

More precisely: Every input port that a forward is connected to, gets two shadow register files of its own. Many forwarding links connected to a single output port cause many write operations and have a negative influence on simulator performance. As an optimisation in the generator an algorithm is used to find out which shadow register files can be shared among input ports. The input ports are grouped by the sets of output ports they are
connected to. If two input ports have the same set of output ports connected, they can share the shadow register files.

The implementation in the generator producing appropriate contributions to the internal description is as follows: The method of the `unit_input_port` class that builds the access environment, inserts an "if" sequence element with a condition comparing the timestamp to the current cycle count. The `build_access_environment()` for unit output ports is more tricky. It has to follow the datapath until it finds the register data port finally written. This lookahead identifies what memory element is to be used to index the shadow register files. It should be noticed that the write access to the forward register file and timestamp registers must be inserted in the epilogue sequence. Otherwise a timestamp from cycle number \( n \) may be overwritten in its successor cycle \( n + 1 \) before it can be checked on the unit input port.

The generated code example shown in Listing 4.4 exemplifies forwarding.

**Listing 4.4: Forwarding in the Generated Code**

```c
void instr_68_stage2 ( union_ops *ops )
{
    fixint <32> EX._ImmWs_i1_1; // Input ports are only local
    fixint <32> EX._Rs_i1_1;

    EX._ImmWs_i1_1 = DE._ImmWs_o1_1; /* move */
    if ( global_time ==
        EX._Rs_i1_1.tstamp_reg [ops->ops_68.R._Rs.getUVal ()]
    ) {
        EX._Rs_i1_1 = EX._Rs_i1_1_forwd_reg [ops->ops_68.R._Rs.getUVal ()];
    } else {
        EX._Rs_i1_1 = DE._Rs_o1_1; /* move */
    }
    ...
}
void instr_68_stage2_epi ( union_ops *ops )
{
    EX._Rs_i1_1_forwd_reg [ops->ops_68.R._Rd.getUVal ()] = EX._Rd_o1_1;
    EX._Rs_i1_1.tstamp_reg [ops->ops_68.R._Rd.getUVal ()] =
        global_time + const_1.64Bit;
    ...
}```
4.3.2 Concurrent Memory Access

If multiple units of a pipeline access memory within a single cycle, a meaningful order has to be defined. This order cannot be given through the xADL description (for now) but is rather implicitly defined: Reads have priority over writes and the pipeline stage determines the priority of a write access (the newest instruction in the pipeline has the highest priority).

VLIW architectures may allow memory read and write accesses within one VLIW instruction. If a read and a write have the same address the old value is read regardless of what slot the read instruction occupies.

To resolve memory access hazards the generator replaces the original memory write by a write to a single shadow register having the size of the memory data port. The shadow register is written back to the actual address in the stage-functions epilogue. The active condition of the original memory would have to be propagated to the epilogue - to simplify programming the propagation of the condition is neglected, instead a timestamp is written (conditionally) that is checked in the epilogue.

Listing 4.5 shows the finally generated C code for a memory write on a MIPS architecture (it is simplified for easier understanding).

```
void instr_34_stage3 (union_ops *ops)
{
  ...
  DCache_ATwriteB = MEM_ST_A_o11; /* move */
  DCache_writeB_shad = DCache_writeB; /* move */
}

void instr_34_stage3_epi (union_ops *ops)
{
  Memory [DCache_ATwriteB.getUVal()] = DCache_writeB_shad.getUVal();
}
```

4.3.3 Concurrent Register Access

In contrast to concurrent memory accesses priorities for register writes are not fixed but can be predefined in the xADL description: The order of the port declaration within the register type definition determines the priority: The first port has the highest priority.

The implementation of this behaviour is as follows: The register file write sequence element
is replaced by a write to a shadow register. The environment of the write access contains
a write to a second shadow register that holds the current highest priority. Both writes
are conditional and only executed if the ports priority is higher than the current highest
priority.

The priority shadow register is initialised with zero and the epilogue only commits the
shadow register to the register file if it sees a non-zero priority. It should be noticed
that the first instruction being executed writes back the shadow register - this is not
necessarily the "winning" instruction. Listing 4.6 shows a simplified generator output for
a conditional PC register write on a MIPS architecture.

<table>
<thead>
<tr>
<th>Listing 4.6: Register Priorities Example: Jump on Overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>void instr_68_stage2(union_ops *ops)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>if ( EX_t._overflow1_1 != const_0_1Bit ) {</td>
</tr>
<tr>
<td>if ( const_127_8Bit &gt;= pc_prior[const_0_32Bit.getUVal()] ) {</td>
</tr>
<tr>
<td>pc_shadow[const_0_32Bit.getUVal()] = EX_.pc_o1_1;</td>
</tr>
<tr>
<td>pc_prior[const_0_32Bit.getUVal()] = const_127_8Bit;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>void instr_68_stage2_epi(union_ops *ops)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>if ( pc_prior[const_0_32Bit.getUVal()] != const_0_8Bit ) {</td>
</tr>
<tr>
<td>pc[const_0_32Bit.getUVal()] = pc_shadow[const_0_32Bit.getUVal()];</td>
</tr>
<tr>
<td>pc_prior[const_0_32Bit.getUVal()] = const_0_8Bit;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

4.3.4 Stalling

As already mentioned hazards can also be resolved by an architecture through stalling
parts of the pipeline. For now stalling is not implemented in the simulator and thus not
considered in the generator. Currently the pipeline is implemented as a single ringbuffer
in the simulator framework but it is imaginable to let the generator produce the pipeline
buffer.

An idea of how stalling could be implemented is to split the pipeline into a variable number
of ringbuffers each having a pipeline stage potentially stalling as its head. Each ringbuffer
would be responsible to fetch its head from the preceding ringbuffer on a new cycle or insert a nop instruction in case of a stall. The execution order could still be reverse inside the ringbuffers beginning with the ringbuffer at the backend of the pipeline. A ringbuffer may only be executed (and advanced), if its successor (behind it in the pipeline) does not stall.

Another way to simulate stalling could involve only a single ring buffer, that - for stall cycles - is advanced by copying entries backwards and insert a nop in the resulting gap after the stalling pipestage-function entry. For advanced pipelines maybe more than one such ring buffer is needed (e.g. one for every subpipeline).

For the interpreting simulator this method could handle "static" stalling (that only depends on the program counter sequence in the pipeline) and "dynamic" stalling (as needed for cache simulation), for the compiling simulator on the other hand (with a simple logic) only static stalling could be handled. Dynamic stalling finally providing cycle accuracy for a generated compiling simulator depends on more research work.

4.4 Further Translations

4.4.1 VLIW Pipeline Stages

VLIW architectures bundle multiple instructions (for simplicity called subinstructions) to a single instruction word. For the adlgen tool and thus the simulator generator subinstructions in VLIW architectures most of the time do not pose a special case. They are found by following the data paths with the only difference that these data paths may encounter duplicated units.

We found it feasible to let the simulator generator see no difference and let it create pipeline stage-functions just like for normal instructions. The only difference is that memory element names get extended by the current slot and the repeat count number (for readability). In the first duplication of unit decode for example the memory element $DEC\_Imm\_i$ would be named $DEC\_Imm\_i1\_4$.

The drawback of this approach is that for identical subinstructions in different slots different pipeline stage-functions are created resulting in lots of duplicated code only varying in memory element names (see evaluation, Section 6.1). On the other hand it was possible to enhance the generator to handle VLIW architectures with only a moderate amount of programming time spent.
4.4.2 Fetch/Decode

The first stages of a pipeline are somehow special in every processor architecture: Their behaviour is independent of the instruction because they operate on an instruction not yet decoded. In simple architectures these stages are fetch and decode, in more complex architectures more stages may be involved (e.g. for instruction unpacking or branch prediction).

Nevertheless it was not clear from the beginning how these stages should be handled in the generator and simulator. At least for now the simulator framework increases the program counter, fetches the instruction word and decodes it by calling a predefined function that is generated (see Section 5.2) and fills the very first stage in the instruction buffer with the decoded information.

Therefore the built-in calls increasing the program counter are filtered out of the architecture description and no precautions for decoding must be taken in the xADL description. All operands may be accessed from the second stage on as the very first stage is assumed to be equal for all instructions (typically at the earliest operands play a role in the decode stage).

4.4.3 Symbolic Values

The simulator framework expects several C symbols to be defined: The program memory pointer and program memory size, the program counter variable, its size in bits and what kind of addresses it holds (e.g. instruction byte address). The data memory pointer and the data memory size are expected to be defined but may be equal to the program memories pointer and size.

Besides that, some structural information is expected, e.g. the length of the pipeline (which is known after iterating over all instructions) and the number of instruction slots (greater than one for VLIW architectures).

The ABI must be respected when handling system calls inside the simulator. As the architecture is assumed to pass parameters and return the handler result via registers it is sufficient for the generator to define appropriate symbols holding register indices (as described in the xADL descriptions ABI section).
Chapter 5

The Interpreting Simulator

This chapter is intended to present the parts still missing to make the interpreting simulator complete. Code snippets have already been used to illustrate concepts of the generator but the details of the generator backend have not been explained yet. The subsumption in a chapter of its own is justified by the fact that for the compiling simulator (presented in [Rigler, 2008]) the backend produces quite different code.

The generated instruction decoder that feeds the pipeline buffer has not been explained yet. It does not use the internal representation and is a rather isolated part of the generator. Nevertheless the decoder is a vital part to get a running simulator and for the interpreting simulator it even poses a performance bottleneck (compare with evaluation, Section 6.3.2).

5.1 Generator Backend

It was already mentioned that the backend of the simulator generator converts the internal description to C code (by invoking polymorph emit_code methods of all objects of the internal description). A parameter that is passed defines what type of code is to be generated. Due to the internal description characteristics the code generation itself is quite simple.

The resulting code for the dynamic translator and concepts behind it are explained in the thesis [Rigler, 2008] while the code generation and the resulting code for the interpreting simulator resulting from different components of the internal description is described in the following sections.
5.1.1 Global Memory Element Pool

Each memory element of the global memory element pool is converted to a global C variable of type fixint. This type is a class template with a selectable bitwidth that the compiler translates to a simple C type (thus for now at most 64 bits are allowed). The fixint class implements several operators and set/get methods to allow convenient usage. Memory-elements marked as constant get the type const fixint.

It should be noticed that all extern definitions for global variables are generated by appropriately invoking the emit_code() functions of memory-elements: The simple call

\[
\text{description.emit_code}(f\_sim\_h, \text{global\_variable\_definition});
\]

is sufficient - it invokes the memory pool emit_code() function which executes emit_code() for every memory element. As this is the lowest level emit_code() method and the parameters are always passed to the next level it can produce the right code.

5.1.2 Operand Memory Element Pool

The operand pools emit_code() method basically must either generate the structs for the union or an access to an operand (as already mentioned a parameter of emit_code() defines the actual behaviour). Listing 5.1 shows the C output for one of many structs within the union.

Listing 5.1: A struct generated for the ops union

```c
typedef struct
{
  fixint_arg <32> R_Rd;
  fixint_arg <32> R_Rs;
  fixint_arg <32> R_Rt;
} operands_66;
```

5.1.3 Pipestage-Functions

For every pipeline stage description the pipestage-function header is written to the target C file. Sequence elements that represent built-in calls are looked up in a table that returns a string with placeholders for arguments and results. The string for the and built-in call for example is:

\[
00 = 00 \& 01;
\]
For condition sequence elements simple if/else code is generated: The if and else parts are filled by calling `emit_code()` for the if and else sequences (for elements of those sequences again the table is used to look up how they translate to C code).

### 5.1.4 Further Backend Tasks

The backend naturally has to produce lots of C glue code like functions to initialise the simulation state and define statements for the symbolic values mentioned above. A field of pointers to stage-functions that is indexed by the instruction number must be initialised - an appropriate loop is generated. Function forward declarations and include statements must also be generated.

### 5.2 Instruction Decoder

#### 5.2.1 Related Work

SLED - the Specification Language for Encoding and Decoding - describes binary representations of machine instructions [Ramsey and Fern, 1997]. A toolkit can be used to generate bit-manipulating code out of SLED specifications. The key contribution is the formalisation of a description (together with a normal form) that enables compact descriptions of both RISC and CICS encodings by introducing the concept of patterns. Patterns describe the concrete binary encoding by dividing bitstreams into tokens and by constraining values of fields. Fields are the lowest level and define groups of bits and their type. Shift and mask operations on fields are hidden to the systems architect, additionally checking mechanisms reduce the possibility of errors (e.g. overlapping fields, gaps between fields).

Two statements connect the description to the process of encoding and decoding: A Constructor is identified by an opcode and operates on operands. The Match statement allows the assignment of host program variables to variables in patterns (so-called bindings). The toolkit creates a decision tree for each Match statement using a heuristic which is unfortunately not described.

The work presented by [Abbaspour and Zhu, 2002] picks up the problem that the porting of binutils (contrary to prevailing opinion) is a hard and time consuming task that is often unavoidable when designing application specific architectures. As a contribution methods to define the syntax and binary representation of an instruction set as well as the relocation and procedure linkage table of the ABI of an architecture are presented. The process of generating and its (working) results are presented for a SPARC instruction set architecture description. The diploma thesis [Mong, 2004] presents the details (e.g.
that the x86 model was given up in favour of simplicity).

In contrast [Baldassin et al., 2007] present a more general description of ISAs that allows
generation of a broader range of processors (results for MIPS, SPARC, PowerPC and
i8051 are presented). Also the generated binutils rely on generated encoding and decoding
routines that could be used for a decoder within a simulator.

An algorithm that creates a decision tree out of a set of binary encodings (with bits being
either 1 or 0 or insignificant) is presented by [Theiling and Absint, 2001]. It is used in
a safety critical environment and thus always evaluates all bits (but each bit only once).
The presented algorithm has a runtime of $n^2 \log n$ with $n$ being the number of binary
entries in the input set.

5.2.2 xADL Binary Encoding Extension

The xADL language allows the definition of multiple binary encodings for the (implicitly)
defined instruction set architecture. In the description for every binary encoding the top
level tag BinaryFormat needs to be defined. A BinaryFormat consists of BinaryTemplate
tags and Bundle tags. Multiple binary formats can be defined for an architecture (but
only one is supported for now).

A binary template represents a whole set of instructions. All instructions sharing the same
position and number of bits for the opcode as well as for the operands (register indices
and immediates) can be covered by defining one binary template. The bits of the opcode
are the significant bits that laterwards identify the concrete instruction. Listing 5.2 shows
the binary template for jump type instructions: The concrete instruction (e.g. jump with
link) is identified by the bits masked in the Mask tag while Token tags represent operands.

Listing 5.2: A Binary Template

```
  <BinaryTemplate name="jtype_b" width="32">
    <Mask name="op" mask="0xfc000000"/>
    <Token name="immj" size="24" pos="0"/>
  </BinaryTemplate>
```

The value of the bit mask for a concrete instruction is assigned in an operation (as shown
in Listing 5.3). The tokens of a binary template (and thus the instructions operands) are
assigned to ports as in Listing 5.4.
5.2.3 VLIW Binary Encoding

VLIW architectures may have a quite advanced binary encoding. For example the CHILI architecture features conditional execution of all (sub)instructions. It uses two slots for a conditional instruction: one for the instruction itself and a second one for the condition. Additionally 32 Bit immediates that capture slots after a whole bundle can be used, meaning that two conditional instructions (in a "worst case") may capture 8 slots in a 4-way CHILI architecture. A reserved bit in each of the four instruction slots signals if that slot demands a trailing 32 Bit immediate and it can be set even if the instruction does not need any immediates.

These enhancements of VLIW instruction encodings have been implemented as special cases: The binary token tag was extended by a placement property that can be set to "trail" and a condition property that can be set to a bitmask selecting the bit demanding for a trailing immediate.

The bundle tags implement a more general concept: They are used to group subinstructions in different slots to a single instruction. The CHILI description uses this to describe the binary encoding of conditional instructions (Listing 5.5 shows a bundle using one slot and another using two slots).

5.2.4 Decoder Generation

For now only a simple "linear" decoder is generated out of the description. An instruction word is fetched and the significant bits are checked in an if/else if structure - every possible instruction is checked in an if clause.
Before the simulator generator produces the decoder, the adlgen tool prepares the binary encoding: The constructor of an instruction tries to match the path of the instruction with a bundle. For non-bundled instructions a bundle will contain only a single binary template - the search for a instruction/bundle matching is trivial, for bundled instructions (e.g. VLIW subinstructions capturing multiple slots) on the other hand a heuristic is implemented that (tries) to find a bundle.

The point is that a single (sub) instruction may have multiple operations (and thus multiple binary templates) on its path. Consider for example a load instruction that uses an add operations output for the memory address and has a second operation to access the memory. Both operations will have a binary template as the add operation is also used by the add instruction.

Therefore non-relevant binary templates on the instruction path can be ignored. The assumption is that always the last operation on the path is the significant one and all others can be dropped. In case of bundled instructions (which have multiple significant and insignificant operations) it must be decided if an encoding can be dropped or if it already belongs to another subinstruction (and corresponds to a significant operation). The pseudo-code of the heuristic is presented in Algorithm 1.

The algorithm is round based. Bundle pointers mark current elements to check in all bundles. All bundle elements pointed to are compared with the current binary template of the instruction (starting with the tail element). If none matches the next binary template of the instruction is selected. Elsewise a new round is started: All bundles with matching elements get their bundle pointers increased and participate in the next round. The heuristic is finished if the current binary template is the first element (head) of the instruction.

The drawback of the heuristic is that in certain cases it cannot decide on a bundle. Consider the following situation: Letters stand for binary templates, insignificant operations are in brackets, the sequence of binary templates is: $A (B) C D$. The candidate bundles are $ACD$ and $EBCD$. Clearly the bundle $ACD$ represents the instruction.

Both candidates survive the first two rounds (for $D$ and $C$) then for $(B)$ only the second
Algorithm 1 Bundle Matching

\[ \text{Algorithm 1 Bundle Matching} \]
\[
\begin{align*}
    \text{bl} & \leftarrow \text{list of all bundles defined} \\
    \text{itv} & \leftarrow \text{pointer to last element of each bundle} \\
    \text{search\_list} & \leftarrow \text{sequence of binary templates used on path} \\
    \text{found\_bundle} & \leftarrow \text{NULL} \\
    \text{for all } \text{element in } \text{search\_list} \text{ beginning with tail do} \\
        \text{for all } \text{bundle in } \text{bl} \text{ do} \\
            \text{if } \text{itv}[\text{bundle}] = \text{element then} \\
                \text{itv}[\text{bundle}] \leftarrow \text{pointer to pred. element in bundle} \\
                \text{add bundle to found\_list} \\
                \text{found} \leftarrow \text{true} \\
            \text{end if} \\
            \text{if } \text{found then} \\
                \text{bl} \leftarrow \text{found\_list} \\
                \text{found} \leftarrow \text{false} \\
            \text{end if} \\
        \text{end for} \\
    \text{end for} \\
\end{align*}
\]
Ensure: \text{bl not empty}

\[ \text{for all } \text{bundle in } \text{bl} \text{ do} \]
\[ \text{if } \text{itv}[\text{bundle}] \text{ points to head then} \]
\[ \text{found\_bundle} \leftarrow \text{bundle} \]
\[ \text{end if} \]
\[ \text{end for} \]

candidate matches. In the final round this candidate does not match and is also removed. To identify ACD as the correct bundle the heuristic would have to backtrace to bring ACD into play again. This weak point of the heuristic has no effect in case of the MIPS and CHILI architectures described but probably needs further investigation.

After the heuristic has identified the bundle the method \texttt{instruction\_t::make\_binary} first fills two byte vectors representing the opcode of an instruction: \texttt{bitmask} has those bits 1 that mask out the opcode of the instruction word and \texttt{bitvalue} holds the actual opcode. Second it creates binary bindings that finally connect tokens to connectables.

The decoder generating part of the simulator generator is kind of an interim solution, its implementation is therefore not presented in detail. The generated top level code can be outlined as follows: First the \texttt{slot\_decode} functions are called for every slot. The returned value is the number of slots actually captured by the decoded subinstruction. Notice that only the first slot is given a (sub) instruction id which is queried laterwards when decoding trailing immediates. Non-VLIW architectures are treated as VLIW architectures with only one slot.
Chapter 6

Evaluation

This chapter presents the quantified results of our work. Besides the properties of the generator and the generated code the runtime behaviour of the interpreting simulator is inspected. Afterwards in the conclusion the most important aspects of our work are highlighted, limitations are pointed out and things not yet implemented (but hopefully in the future) are summarised.

The measurements presented in the following sections have been made on a 2100 MHz AMD Athlon X2 Dual Core Processor BE-2350 with 512 KB cache for each core and 2 GB of RAM running GNU/Linux 2.6.24-2. All results given in MHz advert to this machine. For the measurement of time the user-space value of the \texttt{time} command has been used, except for the generated simulators where the cycle counting TSC register has been used.

6.1 Generator Results

The non-generated part of the simulator (the framework) consists of 3297 lines of code at the time of writing. Depending on the architecture the generator produces code starting from approximately 13000 lines of code (for the MIPS R2000 architecture) up to 780000 (!) lines (for the CHILI architecture) and more. In case of the CHILI this explosion of size is due to the high number of conditional subinstructions: 24 different conditions can be combined with all subinstructions (a special if subinstruction captures one slot and the conditional subinstruction the other).

Table 6.1 shows how long it takes the generator to produce those great amounts of code. Additionally it can be seen that it gets hard for gcc to compile the simulator (all times are user-space times, real-time for gcc is approximately 40\% shorter on the dual-core test system). This issue could be solved by the generator backend eliminating redundant code, but this is left open as future work.

60
### Table 6.1: Generating Times

<table>
<thead>
<tr>
<th>Command</th>
<th>CHILI</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>adlgen</td>
<td>9.91 s</td>
<td>0.16 s</td>
</tr>
<tr>
<td>gcc -O3</td>
<td>36 m</td>
<td>1 m</td>
</tr>
<tr>
<td>gcc -O0 -g3</td>
<td>5 m</td>
<td>37 s</td>
</tr>
</tbody>
</table>

#### 6.2 Simulation Technique and Methodology

The work of [Yi and Lilja, 2006] presents popular simulators and benchmarks and explains techniques that can be applied to gain simulation speed while providing sufficient accuracy (like reducing the input set of a program, statistical sampling and warming). The focus is rather put on the process of simulation than on the implementation of a fast simulator and the evaluation of the simulated architecture rather than the simulator itself.

Also the steps needed to gain meaningful simulation results are described, as simulation results can be so inaccurate that the actual impact of processor architectural changes is obscured. Five of those steps will be reviewed and adopted to the generated simulator (the step where the simulated processors hardware is adopted is omitted).

#### 6.2.1 Simulator Validation and Accuracy

As already mentioned, the generated simulator has some limitations. Stalling is not implemented yet - thus caches (and delays resulting from cache misses) are not simulated and multi-cycle instructions account for only one cycle. The generated simulator could be seen as an instruction set simulator that can be relatively easy extended to support cycle accurate simulations (at least for simple pipelines). How to describe advanced pipelines in xADL and how to generate (cycle accurate) simulators for such pipelines has not been investigated yet and is future work.

Nevertheless the generated simulator has been verified against commercial products. The outputs produced by MiBench benchmark programs run on the generated simulator have been examined and are identical to those produced by the MIPS SDE simulator (an instruction set simulator) in case of MIPS R2000 and identical to those produced by ONDEMANDs hand coded simulator in case of the CHILI architecture.

The instruction counts have been found to be identical: A minor difference of less than 0.1 per mill is measured for the MIPS architecture (which is assumed to originate from the implemented system call handling), for the CHILI architecture instruction counts are exact (the generated interpreter always counts 4 cycles more than the reference simulator instruction count due to initial pipeline filling).
6.2.2 Processor and Memory Parameter Values

This step in the simulation process is of importance if advanced architectures are simulated. It can be challenging to find out which parameters are significant and which are not within advanced architectures. Also parameters depend on each other: The memory latency and bandwidth for example influence the ideal parameters for a cache. For now only "structural parameters" are translated by the generator (e.g. number of pipeline stages or number of slots for VLIW), others (like memory delays) have not been recognised until now.

Basis for all CHILI architecture simulations was the 4-way VLIW type with 8 MB program memory and 16 MB data memory, basis for all MIPS simulations an R2000 derivate with an 8 MB single memory.

6.2.3 Selected Benchmarks and Input Sets

The main focus of the xADL is to describe embedded processors. A benchmark suite that is intended for the evaluation of such architectures is the MiBench suite [Guthaus et al., 2001]. It is therefore the benchmark of choice for our simulator. The MiBench consists of several (rather small) programs covering different domains of embedded systems.

For the simulation a subset of MiBench has been chosen (same as in [Rigler, 2008]): Non-used programs in Table 6.2 are grey (adapted from [Guthaus et al., 2001]). Not all programs can be simulated for now, as the generated MIPS and CHILI simulators still have bugs. It is assumed that at least some of them originating from wrongly described operations in the xADL description. One additional benchmark not from the MiBench is used: The very first program that has been run by the generated simulator, a very simple prime number generator.

All MIPS benchmarks have been compiled with gcc 3.4.6 and binutils 2.18, for the CHILI benchmarks an experimental gcc 4.2.0 (20060617) and binutils 2.16 have been used. Optimisation when compiling benchmarks always has been -O.

The input set for each benchmark (if needed) consists of a single file, that is chosen in respect to runtime (no more than approx. 1000 megacycles) either from MiBench "large" or "small" input files. Table 6.3 shows the input sets chosen for MIPS and CHILI and shortcuts used for the benchmarks later on.

6.2.4 Simulation

A simulation run is supported by a script that reads out a textual table. This table specifies the benchmarks to be run and their input files. As already said, the input set was not
artificially reduced. Furthermore the generated simulator does neither support sampling (where properties of the architecture are statistically deduced) or fast-forwarding/direct execution (where the benchmarks partly run on the host machine) nor it does support warming (where detailed simulation of interesting parts of a program is preceded by a warm-up phase that brings caches etc. to reasonable correct states).

The strategy of the generated simulator is straight forward: Simulate the program as a whole in detail, with the complete input set until it is finished (or a certain amount of cycles is reached). To speed up the process of simulation, the detailed simulation itself is sped up by dynamic translation [Rigler, 2008]. This method guarantees a high degree of precision and may even turn out to be applicable for the simulation of more advanced micro architectures (future work).
6.3 Simulator Results

6.3.1 Simulation Speed

A first run of the benchmark by both generated simulators reveals the ranges of simulation speeds for the interpreting simulator (compare with Table 6.4). While the MIPS simulator runs at a few MHz, the CHILI simulator with between 600 and 800 KHz approximately runs at a fifth (accounting for the 4-slot simulation and slightly more data manipulations within the pipestage-functions). The bitcount benchmark is that short, that loading and initialisation overhead dominate the measurement and dampen the instruction frequency.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MIPS Instr.</th>
<th>CHILI Instr.</th>
<th>MIPS MHz</th>
<th>CHILI MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>prime</td>
<td>20337289</td>
<td>435631063</td>
<td>3.26</td>
<td>0.64</td>
</tr>
<tr>
<td>jpeg</td>
<td>36150283</td>
<td>17497353</td>
<td>3.34</td>
<td>0.73</td>
</tr>
<tr>
<td>crc32</td>
<td>698600505</td>
<td>1389091412</td>
<td>3.26</td>
<td>0.74</td>
</tr>
<tr>
<td>sha</td>
<td>162016520</td>
<td>275919553</td>
<td>3.26</td>
<td>0.72</td>
</tr>
<tr>
<td>dijkstra</td>
<td>311920661</td>
<td>1185939043</td>
<td>3.35</td>
<td>0.73</td>
</tr>
<tr>
<td>bitcount</td>
<td>48227</td>
<td>42197</td>
<td>0.92</td>
<td>0.48</td>
</tr>
<tr>
<td>blowfish</td>
<td>14876929</td>
<td>28269414</td>
<td>3.27</td>
<td>0.61</td>
</tr>
<tr>
<td>stringsearch</td>
<td>6849113</td>
<td>11513913</td>
<td>3.16</td>
<td>0.61</td>
</tr>
<tr>
<td>adpcm</td>
<td>26618847</td>
<td>51284759</td>
<td>3.23</td>
<td>0.66</td>
</tr>
<tr>
<td>gsm</td>
<td>109550686</td>
<td>136257055</td>
<td>3.16</td>
<td>0.70</td>
</tr>
<tr>
<td>rijndael</td>
<td>32618788</td>
<td>60995464</td>
<td>3.20</td>
<td>0.73</td>
</tr>
</tbody>
</table>

Table 6.4: Benchmarks: Instructions, Speed

While the simulation speed for a single MIPS instruction and a CHILI subinstruction seems to be similar, the CHILI benchmark consumes approximately twice as much instruction cycles compared to MIPS. This is very counter-intuitive as a VLIW instruction set is expected to reduce instruction counts and is assumed to be caused by the gcc compiler producing poor code. Fortunately the compiling simulator compensates this and reaches astoundingly short simulation times (presented in [Rigler, 2008]).

The generated interpreting simulator reaches 5 times the clock speed of chilisim, a commercial product to simulate the CHILI architecture (that reaches an average clock speed of 0.12 MHz). The chilisim simulator is cycle accurate but to be comparable to the current version of the generated simulator all cache and stall simulations have been turned off. Compared to sde-run a commercial instruction set simulator for the MIPS architecture, the generated simulator is 40% slower (Figure 6.1).
6.3.2 Simulation Time Breakdown

The simulation time of each benchmark can be broken down to three main categories: time spent in the framework, time for decoding the instruction words from their binary form and time consumed by simulating the instructions in the stage-functions.

![Figure 6.2: MIPS Breakdown](image)

Figure 6.2 and Figure 6.3 illustrate percentages of time spent in each category. On average the decoder consumes 15% of simulation time in case of the MIPS and 10% in case of the CHILI architecture which in turn spends more time in the stage-functions. Bitcount is the only benchmark with a significant amount of time spent in the framework, which is due to its very short runtime. It should be noticed that the pipeline shifting and array accesses account for framework time. It is noticeable that only little time is consumed by the decoder compared to the stage-functions which is due to many redundant copy
operations simulated in the stagefunctions (e.g. read in of input ports and write back of output ports). Those operations could be eliminated in future work to increase interpreter performance.

The generated interpretive simulator was the first step towards the dynamic binary translating simulator and thus performance optimisations for the interpretive simulator have not been an issue.

**Figure 6.3: CHILI Breakdown**
Conclusion & Future Work

Starting with the various aspects of today’s computer architectures, several architecture description languages from related work have been inspected to finally account for the xADL language and the basic concepts of our simulator generator (like its internal description). The correct simulation of hazards (more exact: the methods simulated that avoid them) revealed as one of the most challenging tasks during implementation and is yet not complete. The cycle accurate simulation of stalling logics is left open as future work but assumed to fit well into our generator.

Other things yet to be investigated include a more generic (fully generated) handling of syscalls, the simulation of advanced pipelines (multiple parallel paths) and the generation of full featured binary utils (including evidence that the current description of binary encodings is adequate for a wider range of instruction sets).

Instruction sets with many conditional operations (like the CHILI instruction set) result in huge amounts of code produced. The generator runtime, but more important, the compilation time for the generated code could be shortened by methods eliminating redundant code.

Future work could also deal with the interpreting simulator potentials not yet utilised: Speed-ups may be gained by eliminating redundant operations within stagefunctions. This would shift the performance bottleneck to the decoder that in turn could be sped up by a tree-search or similar methods. Additionally the operand structures of already decoded instructions could be cached.

Nevertheless the work presented supports the claim that generation of cycle accurate simulators out of structural processor architecture description languages is feasible. With the current implementation the corner stone for a solid simulation framework has been laid. Together with the ongoing efforts to generate highly optimising compilers [Brandner et al., 2007] and VHDL code out of the same description, a full processor architecture development toolchain comes into reach. The generation of binutils is the lacking piece of the puzzle but related work puts this within the realms of possibility.

The chosen approach to fully simulate an architecture at RT-level letting it execute the whole programs targeted at it even seems applicable if complete input sets are used as
long as this "simulate everything“ approach draws upon the heavy usage of runtime compilation. The llvm compiler framework seems promising for both runtime optimisations in generated simulators [Rigler, 2008] as well as for generated compilers [Brandner et al., 2007].

The adlgen framework, the generator module and the generated simulator emerged to be handy tools. Together with the fact that the xADL description is very compact (1054 lines for the MIPS description and only 1650 lines for the CHILI 4-way VLIW description) the current xADL approach is primely promising.
Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Wien, am 30. September 2008

Andreas Fellnhofer
Bibliography


