TU WIEN Informatics

Instruction Set Simulation of VLIW Architectures in VADL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Niklas Mischkulnig, Bsc.
Matrikelnummer 11809607

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Wien, 30. April 2025

Niklas Mischkulnig	Andreas Krall

TU WIEN Informatics

Instruction Set Simulation of VLIW Architectures in VADL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Niklas Mischkulnig, Bsc.
Registration Number 11809607

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

Vienna, April 30, 2025

Niklas Mischkulnig	Andreas Krall

Erklärung zur Verfassung der Arbeit

Niklas Mischkulnig, Bsc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. April 2025

Niklas Mischkulnig

Kurzfassung

Von Hand ist das Entwickeln und Testen von neuartigen Prozessorarchitekturen ein zeitaufwendiger Prozess, da viele Instruktionen in vielen einzelnen Programmen implementiert werden müssen. Prozessorbeschreibungssprachen machen diese Arbeit sowohl weniger fehleranfällig als auch effektiver, sodass es möglich wird, sich einer guten Prozessorarchitektur schnell anzunähern. Die Vienna Architecture Description Language (VADL) ist eine solche Prozessorbeschreibungssprache. Very Large Instruction Words (VLIW) ist ein vor allem bei spezialisierten digitalen Signalprozessoren verwendetes Designmuster, bei dem mehrere Instruktionen in ein parallel ausgeführtes Paket zusammengefasst werden.

Bisher unterstützte VADL keine VLIW Architekturen, da es nicht möglich war, gültige Instruktionspakete zu beschreiben. Abgesehen davon war es außerdem nicht möglich zusätzliche (willkürliche) Bedingungen aufzulisten (beispielsweise um das gleichzeitige Schreiben auf ein einzelnes Register zu verhindern). Schließlich war VADL auch nicht in der Lage, einen funktionalen Simulator für VLIW Architekturen zu erzeugen.

In dieser Arbeit erweiterten wir die VADL Sprache, um gültige VLIW Pakete spezifizieren zu können: einerseits mit regulären Ausdrücken (mit zusätzlich Permutationen und Wiederholungen) und andererseits mit Bedingungen auf bestimmte Formatfelder von Instruktionen. Diese Information wurde dann verwendet, um in VADLs bestehendem funktionalen Simulator auch VLIW Instruktionen zu unterstützen. Zur Laufzeit wird das Programm gegen den regulären Ausdruck mithilfe eines deterministischen Zähler-Automaten ausgewertet, wodurch bei der Verwendung von Permutationen und Wiederholungen ein exponentiell großer Automat vermieden wird. Alle diese Funktionen wurden anhand von zwei VLIW Architekturen evaluiert (TI C6x und Qualcomm Hexagon), die (teilweise) in VADL spezifiziert und als Simulator getestet wurden.

Abstract

Developing and prototyping novel processor architectures is a laborious task when done completely by hand, as there are many instructions to implement across many tools. Processor description languages make this work both less error prone and faster to iterate on, enabling rapid design space exploration. The Vienna Architecture Description Language (VADL) is such a language. Very Large Instruction Words (VLIW) is a design paradigm predominantly used by specialized digital signal processors, which groups instructions together into bundles which execute atomically in parallel.

Thus far, VADL hasn't supported VLIW architectures, as it was not possible to specify valid groupings of instruction types. Orthogonally to that, it also wasn't possible to have additional (arbitrary) constraints for these groups (for example to disallow concurrent writes to the same register). Furthermore, VADL's existing instruction set simulator generator also wasn't able to emit VLIW simulators.

In this work, we extended the VADL language to be able to specify valid VLIW bundles with a regular expression syntax (which additionally allows permutations and repetitions) and also arbitrary constraints to constrain specific fields of the instructions. This information was then leveraged to extend VADL's existing simulator capabilities to support VLIW architecture simulation. At runtime, the specified regular expression is matched against the instruction stream using a deterministic automaton with counters which prevents an exponential blowup when using permutations and repetitions. All added features were then evaluated by first specifying (parts of) the TI C6x and Qualcomm Hexagon architectures in VADL, and finally generating a simulator from the VADL Hexagon specification.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

In the embedded computing landscape, it is essential to design processors with costs and performance characteristics which are viable for the specific application. This poses various challenges:

- multiple software artifacts are needed (such as compilers and simulators) to facilitate evaluation and design space exploration,
- specifying a complete architecture is very repetitive (there are many instructions that only differ slightly),
- resulting in very large projects (containing at least hundreds of instructions with dozens of variants each).

Specification languages specifically designed for this use case (domain specific languages, DSLs) offer a way to solve these problems: processor description languages (PDLs, or architecture description languages, ADLs) can be used in the design of new processor architectures to automatically generate the tooling (such as compilers, assemblers and simulators) required for testing and using these new architectures. They usually cover the instruction set architecture (ISA) and in some cases also the implementation of this instruction set in hardware (the micro-architecture, MiA). The Vienna Architecture Description Language (VADL) is such a PDL which covers both aspects.

PDL-based processor development is particularly useful for processors which are tailored to a specific use case in a specific project (application specific instruction set processor, ASIP). In these situations where various processor designs are evaluated for the specific application, automatically generating compilers and simulators (to compile and execute test or benchmark programs) speeds up the development cycle and the design space exploration.

Digital signal processors (DSPs) are a common processor type that is neither an ASIP nor a general purpose processor (as the ones used in most consumer devices). They are often used for fixed workloads, such as accelerating AI computations but also processing audio or RF signals, which require a high and consistent throughput and/or latency. Apart from having tailored instructions which are useful for matrix operations or Fourier transformations, many DSP processors are using a Very Large Instruction Word (VLIW) architecture. The more common superscalar architectures (used in modern consumer CPUs) determine which instructions can be executed in parallel at runtime in hardware. Conversely, VLIW is an approach where the compiler determines this ahead of time and groups instructions together into VLIW bundles (which contain multiple instructions that are executed in parallel). This can have a performance and resource consumption benefit for non-general-purpose computations.

Designing VLIW architectures in PDLs such as VADL is thus only possible if the language has a way of specifying which instructions can be grouped together into VLIW bundles. This information can then be used by the compiler to generate such bundles and by the simulators to execute programs.

Most VLIW architectures allow at most one instruction to use a given resource (such as functional units, each of which can perform one arithmetic operation or memory access) per cycle, requiring the ISA to specify how many such resources are available. Inside the VLIW bundle, these instructions can potentially have an arbitrary order, which cannot be specified with regular expressions alone (other than listing out all $n!$ possibilities where n is the number of instructions per bundle). Often, there are also additional constraints which relate to the operands of the instructions themselves (e.g. a register can only be written to by at most one instruction per bundle).

1.2 Research Questions

RQ1: *How can VLIW bundles be described declaratively?*

Our idea is to use regular expressions as an already well-known mechanism for declarative specifications, while extending them with additional syntax elements to more easily (and efficiently) describe repetition of elements and unordered permutations. Furthermore, constraints similar to first order logic should be able to constrain both the instruction type as well as the used registers and immediates.

RQ2: *How can this grammar be represented as a parser which is efficient in regard to memory- and/or time-space?*

All standard regular expressions can be transformed into a deterministic finite automaton. Our extension however will require this mechanism to be adapted to use an automaton with counters and constraints which can be turned into a deterministic automaton without exponential blowup.

RQ3: *What is the performance overhead of accurately asserting valid VLIW bundles in a simulator?*

Assuming a correct compilation pipeline, verifying in the simulator that all VLIW bundles are well-formed should not be necessary. However, especially during development of a novel ISA, this ensures that the assembler, compiler and linker are all behaving correctly. Furthermore, the hardware also performs this check (and can trigger an interrupt in such a situation).

1.3 Aim of this Work

This thesis aims to extend the VADL system to support VLIW architectures.

- **Support for VLIW architecture descriptions in VADL** It should be possible to describe the most common existing VLIW ISAs using VADL. This specification should cover all details of the ISA, not only the sufficient grouping constraints but also all necessary constraints, as the description is also intended to eventually allow compiler backend generation.
- **Support for VLIW architectures in the VADL simulator** To accompany the previous goal, these specifications can also be used to generate an instruction set simulator which is able to decode and execute programs of VLIW architectures. The simulator should also ensure the correctness of the executed program, just like the hardware also detects invalid bundles.

1.4 Structure of this Work

Chapter 2 starts by giving an introduction into the VADL language and into the basic concepts of VLIW architectures including three concrete VLIW ISAs that will be used as a running example throughout this work. Chapter 3 presents an overview of the existing work covering processor description languages, VLIW simulators, and automatons. In chapter 4, we describe the approach taken and implementation performed to support VLIW in the VADL language and simulator. Chapter 5 evaluates our changes by specifying some VLIW architectures in the VADL language, and by measuring how the VADL simulator performed compared to existing tools. Finally, chapter 7 concludes this thesis.

CHAPTER 2

Background

This chapter describes the most important aspects of the VADL language (excluding the VLIW support that was introduced in this work and will be described later on), and the fundamentals of VLIW architecture to be able to understand which problems need to be addressed to support VLIW in VADL.

2.1 VADL

VADL is a language to describe processor architectures. It always contains the instruction set architecture which is the processor behavior that users/developers can depend upon (also regarding backwards and forwards compatibility). Additionally, the micro architecture as the (internal) implementation of an ISA can be specified as well. An important design decision was to clearly separate ISA and MiA, both to differentiate the two aspects and also to make it easy to experiment with different MiAs which implement the same ISA during the design phase.

There are multiple backends for the VADL compiler that can generate different artifacts: simulators (both functional and cycle-accurate), a compiler (including assembler and linker), and hardware (Chisel code that can be compiled to Verilog and eventually etched into a silicon chip). This has guided the language design to make it even feasible to generate a compiler (which requires knowledge about the structure of the ISA and the behavior of the instructions to derive the necessary information for instruction selection), or the hardware (which is inherently parallel instead of sequential as most programming languages are).

For this reason, the VADL language itself has a functional style, apart from writes to registers or the memory. These writes are also not visible until after the current instruction has finished executing, which aligns with the parallel execution paradigm of the hardware.

An ISA is declared with **instruction set architecture** which then contains all structural elements and instructions, as shown in Listing 2.1.

There are some resources that model the state of the processor, most importantly registers and the main memory. Listing 2.1 shows an example of an ISA called `DSP` with a byte-addressable 32-bit main memory, 32 general purpose 32-bit registers and a separate 4-bit register for predicates. The other basic datatypes apart from `Bits` are `UInt` and `SInt` which are all arbitrary length bit vectors and integers, respectively. All of these types have the concrete bit width as a type parameter, but type inference allows also using the abstract type when casting values.

The most important register is the program counter. It can have a `next` annotation to denote that it points to the next instruction, as opposed to the instruction that is currently being executed (which is the default behavior). This is a somewhat arbitrary decision from an ISA perspective that is nevertheless required for some ISAs (and is usually done to simplify the microarchitecture implementation).

```
1  instruction set architecture DSP = {
2    memory MEM : Bits<32> -> Bits<8>
3    register file X : Bits<5> -> Bits<32>
4    register PRED: Bits<4>
5
6    [next]
7    program counter PC : Bits<32>
8
9    // ...
10 }
```

Listing 2.1: VADL Resource Declarations: Registers and Memory

Each instruction is composed of multiple aspects: the behavior and the encoding (both immediates like operands and the opcode). As shown in Listing 2.2, there's the instruction format which has a base type (32 bits) and assigns names to some bit ranges (similar to C's packed bit fields). These format variables can then be used in the **encoding** block to specify one or multiple opcode fields. Format variables can also be referenced in the **instruction** body itself, like in the `ADD` instruction which adds the immediate in `rs2` to the register with index `rs1` and stores the result in register `rd`.

Most instructions don't modify the program counter, in which case the program counter is implicitly incremented to point to the next instruction (based on the length of the current instruction).

The instruction `CCALL` in Listing 2.2 shows how a common RISC conditional call instruction might be implemented by setting `X(31)` (the link register) to the return address and incrementing the program counter by a shifted displacement constructed from format variables. It demonstrates basic control flow (based on the predicate register from above) and special methods available for the program counter: `PC.current` refers to the

address of the current instruction and `PC.next` refers to the address of the following instruction (assuming no jumps). For most architectures, these are just syntactic sugar (i.e. a constant offset) since the length of the current instruction is always known (the format of `CCALL` is `F` whose size is `Bits<32>`). However, this will become more complex with VLIW instructions.

```
instruction set architecture DSP = {
  // ...

  format F : Bits<32> =
    { par    [31]
    , rs2    [20..16]
    , rs1    [15..11]
    , rd     [10..6]
    , opcode [5..0]
    }

  instruction ADD : F =
  {
      X(rd) := X(rs1) + rs2
  }
  encoding ADD = { opcode = 0b00'01'00 }

  instruction CCALL : F =
  {
      if PRED != 0 then {
        X(31) := PC.next
        PC := PC.current + ((rs2, rs1, rd) << 2)
      }
  }
  encoding CCALL = { opcode = 0b00'11'00 }
}
```

Listing 2.2: VADL Instruction Declarations

For VLIW, it will prove useful to be able to group instructions into sets, which in VADL are called **operation** (these are also used in the microarchitecture specification). They can be declared with annotations or by listing out the instructions (or other operations for a set union) in the declaration, as shown in Listing 2.3. By themselves, operations have no inherent semantics and only exist to be able to easily refer to multiple instructions in other parts of the language.

```
instruction set architecture DSP = {
  // ...

  [ operation OP_ALU ]
  instruction SUB : F = // ...

  operation OP_ALU       = { ADD }
```

```
8   operation OP_MEMOP     = { }
9   operation OP_ALU_MEMOP = { OP_ALU, OP_MEMOP }
10 }
```

Listing 2.3: VADL Operations

VADL also supports many more language constructs that are not relevant to this work, such as syntactical macros, microarchitecture/pipeline specification, and exceptions. See [Fre+25] for a full description of the language.

Finally for simulator or hardware generation, a `micro processor` declaration is required, which provides initial values for registers and the memory, and specifies a stop condition (which is essential for a simulator to cleanly exit, but does not apply to the compiler or the hardware backend). Listing 2.4 shows a `micro processor` declaration for the `DSP` ISA that was described thus far. It sets an initial value for the program counter and exits automatically once the program counter has a specific value. Upon startup, the stack pointer is initialized (and if there were any, control register initialization would also be done here).

```
1  instruction set architecture DSP = { /* ... */ }
2  micro processor CPU implements DSP = {
3    start = 0x80000000
4    stop  = (PC = 0xeeee'eeee)
5
6    startup -> ( ok : Bool ) =
7    {
8      X(29) := 0x40000000
9      PC := start
10
11     ok := true
12   }
13 }
```

Listing 2.4: VADL Microprocessor Declaration

2.2 VLIW

VLIW (Very Large Instruction Words) is, at its core, an „architectural design philosophy“ ([FFY05, p. 54]). It was developed to increase processor performance (which can be expressed in the number of instructions executed per clock cycle, IPC), just like today's ubiquitous superscalar processors, by leveraging instruction level parallelism (ILP).

The following three sections about VLIW's core principles are based on [FFY05, Chapter 3].

2.2.1 Motivation

Pipelining ILP is crucial to execute one or even more instructions per clock cycle. Executing an instruction fully in a single clock cycle is not possible, so all processors are *pipelined*, meaning that the instruction execution is split up into multiple stages (such as fetch, decode, execute, memory operations, result write-back). In some situations, this allows the instructions to be executed in an overlapping fashion, achieving an IPC of one (though in reality, five stages of five different instructions are executed every cycle). However, once an instruction depends on the result of a preceding instruction that is also still executing, it has to be *stalled* and wait until the result is available. This is called a *data hazard*, caused by the data dependency between the instructions. Similarly, conditional jump instructions also cause *(control) hazards* because their execution result determines which instruction to run next.

Superscalar Superscalar architectures (also called *multiple-issue* architectures) are able to achieve an IPC greater than 1 by issuing (starting execution of) more than one instruction at once. Compared to pipelining by itself, this requires that the processor is able to execute a single stage of multiple instructions per cycle, thus its functional hardware units (like an ALU for arithmetic operations) need to be duplicated multiple times. Hazards prevent leveraging superscalarity just like they do for pipelining itself, but the relative slowdown is even higher, because a theoretical IPC of (e.g) 2 falls to below 1.

Processors based on the concepts introduced thus far would work somewhat well, but the data and control hazards would pose a bottleneck for performance.

OoO and Speculation Therefore, essentially all modern superscalar processors (for high performance, so excluding for example low-power embedded applications) perform *out-of-order (OoO) execution*, which handles data hazards by executing instructions still further ahead in the instruction stream that don't have to wait for a result of a preceding instruction. Control hazards are handled using *speculative execution*, by guessing the result of conditional jumps and then rolling back all changes in case the jump was predicted incorrectly. For this, the processor has to analyze the data dependencies between instructions at runtime to determine how to reorder and where to speculate.

Another perspective onto this approach is that the program (instruction stream) doesn't specify the execution order anymore, but is only a serialization of a graph that represents the computation that should be performed. OoO execution reorders it into an equivalent serialization, while speculation handles the side effects that are necessarily part of the not completely side-effect-free computational graph (notably the jumps, memory writes, and architectural register writes).

VLIW An alternative solution for superscalarity that works without OoO and speculation is to explicitly group instructions without any dependencies together. This way the processor can simply execute one grouping after another without any dependence

analysis while still issuing as many instructions as possible. The dynamic dependence analysis and scheduling is now moved from the processor hardware into software, where the compiler generates a static schedule.

For non-VLIW architectures, the number of functional units is part of the microarchitecture and is thus not exposed in the ISA. In a VLIW processor, this becomes part of the ISA as compilers need to know which units are available when grouping instructions together. Similarly, register bypassing is usually a microarchitectural detail, but some VLIW architectures expose it in the ISA by allowing reading some values which were only computed in the same group.

The benefits of VLIW are that fast processors are less complex (because the static scheduling happens ahead of time, requiring less run time resources) and thus cheaper to produce. For certain workloads, they can be faster overall. On the downside, exposing more microarchitecture in the ISA leads to worse backwards and general compatibility of binaries across processors (both across versions of the same MiA and across different processors with the same ISA). An important aspect to consider is that the way scheduling/grouping is done by the compiler affects VLIW performance much more, as bad scheduling and subsequently many very short groups effectively turn a VLIW processor into a much slower non-superscalar processor.

This shows why VLIW processors are currently only used in embedded applications: lower cost, firmware that doesn't need to be compatible with other devices, DSP usecases that tend to perform well in VLIW architectures.

2.2.2 Terminology

There are no standard terms used in literature and by processor designers for the *instruction unit* (e.g. a single `ADD` instruction) and the resulting *group* of instruction units (e.g. two parallel `ADD` instructions), as the term *instruction* is now ambiguous.

There are two different dimensions to differentiate: the abstract entities in general, and the encoding (which is some bit vector) of these entities.

The abstract *instruction group* (not a meaningful concept in non-VLIW architectures) is called *(issue) group*, *bundle* or *(execute) packet*. The abstract *instruction unit* (what non-VLIW architectures call *instruction*) is sometimes called operation or simply instruction.

An encoded instruction group is sometimes referred to as a *word* or *fetch packet*. An encoded instruction unit as a part of the bundle word might be referred to as a *syllable*. [FFY05, p. 87]

For example, Qualcomm uses *instructions* and *packets* for their Hexagon architecture [Mana]. Texas Instrument's C6x processor series has *instructions* in *fetch/execute packets* [Manc]. Intel Itanium uses the terms *instructions* and *instruction group* (and additionally *bundle* for the subgroup described below in 2.2.3) [Manb].

In this thesis, we use *instruction* for the atomic instruction unit, the remaining terms are unambiguous anyway.

2.2.3 Encoding Schemes

There are multiple high-level approaches to encode a group of instructions into a bit vector:

Fixed-Width An n-issue processor needs at most n instruction in the group. The most straight-forward encoding is a group word which always contains n instructions, with each slot being assigned to a fixed functional unit. This is also the easiest encoding to decode (because of the fixed instruction-to-unit mapping). The major downside, however, is that many groups end up containing many no-op instructions to pad out the unused functional units. This leads to wasted memory and also (if employed) more instruction cache misses and thus higher energy consumption. [FFY05, p. 114]

Figure 2.1: An example for a fixed-width encoding scheme, each group contains five instructions.

Therefore, the following encoding schemes all rely on variable length group words (to prevent excessive no-ops).

Fixed-Overhead/Mask Each word starts with a syllable-sized mask that specifies both the size of the group and the mapping of instruction to functional unit. This way, the decoder can effectively expand the compressed word into the equivalent fixed-width encoding. As the name implies, short words have a proportionally larger overhead. [FFY05, p. 115]

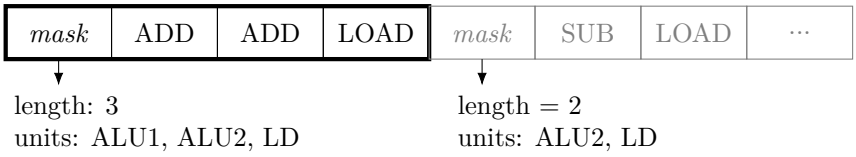


Figure 2.2: An example for a mask-based encoding scheme: two groups with different lengths and what information is encoded in the mask.

Variable-Overhead/Distributed The fixed overhead can be distributed into the syllables themselves: using a start/stop/parallel bit per syllable that specifies whether the start/end of the current bundle was reached (thus encoding the length), and also encoding the appropriate unit in the syllable directly. This approach is used in the TI C6X and Qualcomm Hexagon architecture. [FFY05, pp. 115 sq.]

Figure 2.3: An example for a distributed encoding scheme, a preceding parallel bit indicates if a given instruction is the last instruction in the current group. The function unit is also encoded as part of the instruction (even if this isn't shown in the figure).

Template By encoding the group ending and the unit mapping into a group header, the encoding can be made more flexible: now, multiple subgroups that each have a header can be chained to build the group itself. *Stops* can be at arbitrary positions inside (or at the end of) the subgroups, depending on the chosen template bit encoding. One shortcoming is that this template header doesn't allow all possible combinations of instructions and stops/chaining, so sometimes no-ops still need to be inserted. Intel Itanium uses this encoding. [FFY05, pp. 116 sq.]

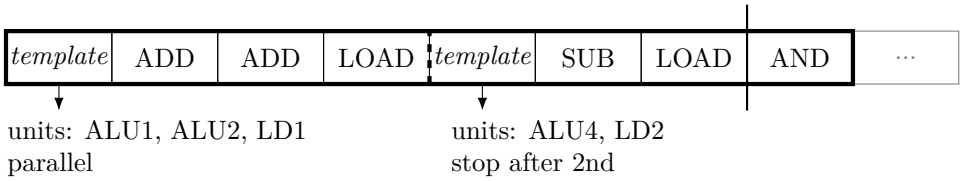


Figure 2.4: An example for a template-based encoding scheme: the two fixed-length subgroups (each with their own functional unit mapping) encode a group of five instructions and another instruction that will be executed as part of the next group.

2.2.4 Examples

After these theoretical foundations, the following (mostly) contemporary VLIW architectures aim to illustrate some ISA features that are unusual compared to the more well-known regular superscalar architectures.

Texas Instruments C6000/C6x

The Texas Instruments TMS320C64x architecture [Mana] (with alternative names *C6000* series or just *C6x*) is a 32-bit DSP architecture targeted at telecommunications and other signal processing applications.

The functional units are split into two subsections (called data paths) in the processor, each containing the same four functional units (multiplication, logical operations, logical operations or branches, memory accesses). Each data path has its own register file with 32 registers, while still allowing one source register read of the opposite register file per cycle via a *cross path*. Because of the separate register files, two functional units of the same type cannot be used interchangeably as they each have access to different registers and thus the unit assignment is part of the instruction encoding.

The architecture has exposed all instruction latencies in the ISA, meaning there are delay slots not only for branches but other instructions as well. Stores and many simple

instructions execute in a single cycle without a delay slot, some instructions such as `DOTP2` (a signed dot product of two packed 16-bit vectors stored in two registers) have three delay slots before the result is written back, loads have four delay slots, and branches have five delay slots. Nevertheless, the functional unit latency is always one cycle, meaning that all instructions can be dispatched continuously on every cycle, just the result is only available with a delay. Due to the explicit delay slots, the MiA doesn't have to handle any hazards and doesn't have to stall the pipeline.

There are *fetch packets* and *execute packets*. Fetch packets are always 256 bits long (eight regular instruction words) and execute packets (containing up to eight instructions) are some subsequence of instructions in the fetch packet (but can still span across two fetch packets) that execute in parallel.

Regular execute packets can contain up to eight instructions in any order (to fill the two data paths with up to four instructions each corresponding to the functional units). They are encoded with a distributed encoding with parallel bits (as shown in figure 2.3).

The C64x+ ISA also supports *compact instructions* which use a template-based encoding that can use shorter 16-bit instruction words for higher code density. In that case, the eighth instruction in the fetch packet is a header specifying which instructions in this packet are encoded in the compact format, where the parallel stop bits inside the packet are, and some other modifiers. Naturally, only a subset of the instructions and operands can be encoded in the 16-bit compact format.

Qualcomm Hexagon

Qualcomm Hexagon [Cod+14; Mana] is a 32-bit DSP architecture used in coprocessors of Qualcomm mobile CPUs for modem and multimedia acceleration. It is a more modern architecture without delay slots and with simultaneous multithreading.

Instructions are partitioned into ten instruction classes, each bundle can then contain up to four instructions that are scheduled into one of four slots, each of which can execute a subset of instruction classes. For example, LD (load) and ST (store) instructions can execute in slot 0 and 1, while JR (jump with register) instructions can execute only in slot 2. The order of instructions in a bundle matters (in reverse order corresponding to the slots) and in case a bundle contains fewer than four instructions, a no-op is implicitly inserted for the slot that remains empty. The bundles use a distributed encoding with parallel bits.

Hexagon's approach for a compressed encoding are *duplex instructions*, which are pairs of instructions that are encoded as a single 32-bit word. A bundle can contain at most one duplex instruction, which must also be the last instruction in the bundle. As always, only a specific subset of instructions, combinations and operands are allowed as duplex instructions.

There are *constant extenders* which are encoded as a 32-bit prefix to enable instructions with full 32-bit immediates (26 bits in the extender plus 6 bits in the instruction itself)

for a subset of instructions, which makes the ISA effectively variable length. Bundles can, however, still only contain four words, so for example either four instructions or three instructions with one constant extender.

The two parallel bits in each instruction are also used for hardware loop support to mark the start and end of the loop body. The loop is then executed for a specified number of iterations without having to include any compare or branch instructions at the end of the loop body (see listing 2.5).

```
    loop0(start,#3)
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Listing 2.5: Hexagon Hardware Loop Example which executes a multiplication three times without any compare or branch instructions.

There are multiple types of instructions that seemingly defy the usual VLIW approach of executing all instructions in parallel (with results only being available in the next cycle).

Dual stores occur when a bundle contains two memory store instructions, these are then defined to be executed sequentially. Similarly, *dual jumps* occur when a bundle contains two jump instructions, which allows constructs such as `if ... jump X else jump Y` or `if ... jump X else if ... jump Y`.

The other type of sequential execution (or rather result forwarding exposed in the ISA) are *new-value* and *dot-new* operations (which use a `.new` operand suffix in the assembly): *new-value stores* allow storing a value that was computed in the same bundle (and therefore same cycle), while *new-value compare jumps* allow conditional jumps depending on a value that was computed in the same bundle. Listing 2.6 contains an example for both of these instructions. Only these two specific types of instructions support new-value operands. Listing 2.6 also shows how predicate registers can be written and read in the same cycle.

```
{
    R0 = R4 + #8
    memw(R5) = R0.new
}
{
    R0 = memw(R4 + #8)
    if (cmp.eq(R0.new,#0)) jump:nt target
}
{
    P0 = cmp.eq(R2,#4)
    if (P0.new) R3 = memw(R4)
    if (!P0.new) R5 = #5
}
```

Listing 2.6: Hexagon example of new-value operations for storing a newly computed value, branching based on a newly computed value, and using a newly computed predicate.

Finally, multiple compare instructions writing to the same predicate register in a single cycle automatically AND the results together, as shown in listing 2.7.

```
{
    P0 = cmp.eq(R0, #2)
    P0 = cmp.eq(R1, #7)
    if (P0.new) jump:t target
}
```

Listing 2.7: Hexagon example for new-Value predicates and auto-AND predicates that performs a jump if two equality checks evaluate to true.

Intel Itanium

Intel's Itanium architecture (*IA-64*), was developed as the 64-bit RISC successor to Intel's X86/IA-32 CISC architecture. It was the result of a partnership with HP (and their Explicitly Parallel Instruction Computing *EPIC* architecture) [Lei19].

It uses a template based encoding as shown in figure 2.4. A 128-bit bundle contains three 41-bit instructions and a 5-bit template. The realized microarchitectures only ever supported executing two bundles (six instructions) in parallel, but this approach would have allowed more powerful processors with more parallelism in the future (without having to recompile any code). The 5-bit template can contain one of 24 different combinations of slot-to-unit assignments and stop positions (plus eight reserved values). Two of the templates can contain an extended instruction, which consists of a 32-bit instruction with a 32-bit suffix to achieve a long immediate integer or branch target.

There are 128 64-bit registers, of which 96 are *stack registers*. They are stored and restored transparently onto or from main memory when performing procedure calls when running out of registers. This is used instead of the more common (e.g. with x86/ARM) approach of always explicitly saving registers onto a stack in memory.

As opposed to the other architectures, Itanium also has a focus on speculation in addition to the VLIW-based ILP. There are *advanced load* instructions which are similar to load-reserved instructions (but without multithreading) which load a value in advance and then potentially re-execute code that depended on the value in case it was outdated. [Manb][Dan, pp. 97-120]

Unlike C6x and Hexagon, the Itanium architecture was discontinued by Intel and the last processors were shipped in 2021 [Arsa]. Even before that, Microsoft decided to stop supporting Itanium in future versions of their server software in 2010 [Arsb]. One of the

reasons for this failure was that the Itanium compiler did not produce good enough code to take advantage of the architecture with its VLIW and speculation aspects. As opposed to fixed-workload DSP applications, general purpose workloads made unpredictable and variable latency branches and memory accesses hard to schedule while still keeping the VLIW pipeline filled. As a competing architecture, AMD's 64-bit x86 efforts had more backwards-compatibility and multithreaded processors lead to more performance gains without the technical complications of Itanium. [Lei19]

CHAPTER 3

Related Work

3.1 PDLs

ISDL [HHD00] is a PDL specifically designed for VLIW architectures. Like VADL, it also supports automatically generating an assembler, instruction set simulator, and code generator. VLIW bundle constraints are described using a language for temporal logic predicates, thus being more expressive than VADL's regular expression approach. The simulator performs static binary translation ahead-of-time, meaning the decode speed does not influence the simulator's performance.

ISADL [XL23] focuses on automatically generating an instruction encoding based on requirements (number of functional units, list of encoding fields included in all instructions, list of encoding fields used by some subset of instructions), with the goal of generating an optimal instruction encoding (regarding binary size) for VLIW architectures. Instruction semantics or constraints are not covered by the language itself.

TDL [Käs03] is not specifically a PDL, but designed for generating optimizers to run on some compiler output. The specification is split into three sections: resources such as registers and memory (and notably, also functional units, unlike in VADL), instructions, and constraints. It allows specifying VLIW constraints in a very similar way as VADL (see section 4.2), using boolean expressions with some special operators to check for membership of an instruction in an operation set.

In nML [FVF95], instructions are described using hierarchical top-down composition of alternatives parts (*OR*) or parallel parts (*AND*). In this model, VLIW bundles themselves could be described as instructions without the need to separately specify constraints. Arbitrary additional constraints that go beyond functional unit availability can, however, not be modeled like this.

LISA [Pee+99] is a PDL designed to be able to generate cycle-accurate models of DSP architectures (specifically the TI C6x architecture). LISA is similar to VADL in that

there are resource definitions for registers and memories, and then each instruction has an assembly encoding, binary encoding and behavior. The root instruction is declared explicitly using an unnamed instruction which can contain one or more sub-instructions, with the same syntax used for declaring the source and destination registers of instructions. The same mechanism can also be used to declare a (pseudo) instruction representing some set of instructions.

[Bra+04] presents an extension of LISA for replacing its C-based behavior definition with more restricted assignments, that also allows composing operations together (analogous to bundles). Multiple VLIW architectures were modeled with this language, but the paper doesn't describe how constraints were handled.

[Res+03] presents a retargetable simulator which has *instructions* (bundles) with *slots* for *operations* (instructions) and also *operation classes* to group instructions. While VLIW architectures are listed as target architecture for this simulator generator, no such architectures were used in the evaluation (neither regarding performance nor expressiveness).

[P+11] describes an extension of the ISAC language (which itself is based on LISA) for VLIW architectures and multi-core processors. Bundle decoding and encoding is specified with a manual implementation using a subset of C, making it harder to automatically generate optimal decode trees for both the simulator and a hardware implementation (using VHDL). The paper mentions that there are predefined bundle compression types (e.g. distributed or template-based encoding) and a way to succinctly define multiple clusters (data paths), but doesn't go into detail how these are specified in the language.

Hewlett-Packard's PICO system not only generates an assembler, compiler, simulator and hardware implementation of a described VLIW processor, but also automatically finds the most cost-effective implementation of a system composed of the VLIW processor, a co-processor used for acceleration, and a cache subsystem. It modeled constraints as exclusion groups (whose contents cannot be executed in parallel, similar to functional units), which can contain both instructions and other instruction groups. The system is based on templated encoding (which was developed for HP's EPIC architecture which turned into Intel Itanium) [ARRK99] [ARR00]. [AMR00] used the PICO system for design-space exploration to demonstrate how adding more templates can reduce code size.

3.2 VLIW Simulators

[HPF13] uses static binary translation (with manually implemented ISA frontends) to simulate the TI C6x architecture by leveraging the Linux kernel's KVM interface for acceleration. The research group's earlier paper [MFP12] describes how register allocation was used to simulate parallel execution by employing register renaming and how delay slots were translated.

[BBR00] describes a simulator which performs, among other things, TI C6x decoding and also pipeline simulation using a reservation table. Parallel execution semantics are modelled by reading from one register files while writing to another separate register file, so that writes within the same bundle aren't visible until the next machine cycle.

[ZHS10] implements an instruction set simulator for TI C6x using queues, which trigger the individual pipeline stages after the correct number of machine cycles (the latency) has passed. Parallel execution semantics are observed as the write back stage is only executed once all operands are already read.

[WYW08] is a simulation framework for VLIW architecture evaluation (consisting of an assembler and simulator), the paper doesn't describe how the VLIW architecture itself is described or parameterized. The simulator has two different execution modes: a purely functional instruction set simulator, and a cycle-approximate simulator emulating the microarchitecture.

[RGD06] introduced using colored petri nets for cycle-accurate VLIW simulation by modelling the execution pipeline as a petri net with individual tokens (instructions) progressively moving through the stages. [Yan+16] leverages colored petri nets for implementing a cycle-approximate VLIW simulator on top of the gem5 simulator.

[Mor+97] describes a framework developed by IBM and consisting of a VLIW compiler and binary translator (outputting PowerPC binaries), which can also include instrumentation for cycle counting. It was designed for the ForestaPC architecture, which has tree instructions instead of regular VLIW bundles. Like bundles, all operations in a tree instruction are also executed in parallel, but the tree can have conditional branches allowing predicated execution.

Tree instructions can be decomposed at runtime depending on the processor's execution width. This approach can have less than a 10% runtime performance overhead compared to compiling for a specific VLIW width while still allowing a single binary to be used with different VLIW microarchitectures [MM97].

[Liu+05] also uses tree instructions but includes differential encoding which is able to more efficiently compress instructions by referencing back to an encoded instruction and re-instantiating it with different operands. Across multiple benchmarks, this reduced the program size by around 64%.

3.3 Automatons

The most well known deterministic finite automaton (DFA) construction method is Thompson's construction [Tho68] (which recursively builds an automaton composed of small building blocks) followed by determinization using powerset construction [RS59] (which simulates all reachable combinations of non-deterministic states).

Glushkov's construction algorithm [Glu61] creates an ε-transition-free non-deterministic finite automaton (NFA). Each symbol in the regular expression becomes a state, and

the transitions are then defined using the symbol's $first$, $next$ and $last$ sets (similar to the ones in parser theory)

Brozozowski [Brz64] introduced *derivatives* of regular expressions (a $u^{-1}L$ which gives all words in the language L starting with the string u with prefix u removed). This notion can then be used to directly generate DFAs from regular expressions. Based on Brozozowski's derivates, Antimirov [Ant96] used a similar approach called *partial derivates* to instead generate NFAs from regular expressions.

Another method that will be used later in this work are *pointed regular expressions* [ACT10] which can be used to directly generate DFAs. Section 4.3.2 will describe this approach in more detail.

A common extension of the normal regular expressions use *permutations* or *unordered concatenation* (being able to specify that multiple symbols have to be matched, but in any order), and *repetition* or *numerical constraints* (being able to specify that a symbol has to matched at least n but at most m times). This class of regular expressions are generally exponentially more concise while still describing the same language as normal regular expressions [Hov12]. However, regular expressions with permutations already make the membership problem NP-hard (that is, even without repetitions) [Hov12].

Some schema languages such as SGML and XML use a different set of regular expressions, supporting both more features (permutation and repetition) while only allowing a subset of the usual regular expressions (*1-unambiguous languages*, meaning that all words can be matched in only one way by the regular expression, and with only an one-symbol lookahead) [BKW98].

[Hov09] developed a method to construct DFAs with counters for regular expressions with repetition. It only works for a subclass of regular expressions (*counter-1-unambiguous* expressions which follow the exact same reasoning as the 1-unambiguity from above, only extended for this superclass of counter automatons) but constructs a DFA in polynomial time while also being able to determine in polynomial time whether the regular expression input is within the supported subclass.

[BC08] also uses counters to support repetitions and back-references. It acknowledges that the number of active counters needed during matching is not constant (which is an alternative solution to restricting the algorithm to counter unambiguous languages only), instead allocating more counters as necessary.

[Tur+20] also generates counter-automatons from regular expressions with repetition, first by constructing a nondeterministic automaton based on Antimirov's partial derivates, and then determinizing them in a second step. Only a subset of regular expressions can be accurately represented because this approach can result in an over-approximation of the original regular expression (i.e. the automaton matches a word that is not matched by the regular expression, so called *non-uniformity*, which is once again related to the problem of counter ambiguity).

CHAPTER 4

Implementation

This chapter discusses the main contributions of this work. Section 4.1 and section 4.2 describe how VLIW support was added to the existing VADL language for non-VLIW architectures (as described in section 2.1). Section 4.3 presents how group expressions are turned into an automaton. Finally, section 4.4 outlines the constraint implementation and section 4.5 describes how correct bundle execution semantics are ensured.

4.1 General Approach

One core goal of VADL is to support generating multiple artifacts, such as simulators and compilers, from the same specification. This necessitates a declarative specification that is also high-level enough to not be too specific to a single artifact.

VADL already supports regular (superscalar) instructions and instruction sets. These constructs should be reused for the instructions in the VLIW groups by composing them into groups by specifying which sequences of instructions are valid.

We chose to start with adding support for fixed-width and distributed encodings, which are used by the Hexagon and C6x architectures.

4.2 Language Design

There are two independent fundamental aspects influencing whether a given group is valid: functional unit availability (e.g. number of ALUs, number of memory ports), and preventing parallel writes. Additionally, the instructions might also have to be in a specific order (e.g. first ALU operations, then followed by memory operations), which can simplify the decoder implementation. (There might be more constraints imposed by an ISA which don't fit into one of these two aspects, usually because constraints imposed

by the MiA have to become part of the ISA, which includes arbitrary design decisions made to improve performance or lower production costs).

Functional unit availability and order are described in VADL with an extended version of regular expressions. Data dependencies and all other constraints which cannot (easily) be expressed in the regular expressions are described in logical constraints using a version of first-order logic. This is also how the stop/parallel bits of the distributed encoding are specified.

Regular Expressions

The extended regular expressions are specified in a **group** definition (as shown in listing 4.1). Any **instruction set** may contain at most one **group** definition. The name of the definition is irrelevant for the regular expression, but will be used in the constraints (see below).

As opposed to normal regular expressions which operate on strings of characters, these operate on instructions and operations (which are sets of instructions). At runtime, a sequence of instructions is matched against the regular expression to determine if it is a valid group. Instructions are, however, not used in the regular expression syntax: instead operations are the terminals in the syntax. They have the semantics of matching any instruction that is contained in the operation.

The syntax itself (specified in listing 4.2) is very similar to standard regular expressions: there are the operation *literals*, *alternatives* (expressions separated by a vertical bar) and *sequences* (expressions separated by a dot). The extensions are *repetitions* of literals by specifying a range in angle brackets after a literal (such as `A<2..4>`, which means „at least 2 and at most 4 repetitions of any instruction contained in operation A“), the lower end of the range can also be 0. The other extension are *permutations* that use curly brackets, which match all listed subexpressions but in any order.

Listing 4.1 shows some examples of group definitions (which are all possible but not necessarily practical as real-world VLIW encodings).

```
1  instruction set architecture DSP = {
2    // ...
3
4    operation A  = {ADD, SUB}
5    operation LD = {LOAD, STORE}
6    operation J  = {JUMP, JUMP_COND}
7
8    group VLIW = (A.LD)         // A followed by LD, meaning
9                                // (ADD or SUB) followed by (LOAD or STORE)
10
11   group VLIW = (J | A | LD)   // J or A or LD
12   group VLIW = {A, LD}        // (A followed by LD) or (LD followed by A)
13
14   // 0-2 As followed by optional LD followed by optional J
```

```
15   group VLIW = ( A<0..2>.LD<0..1>.J<0..1> )
16   // as previous group, but any order
17   group VLIW = {A<0..2>, LD<0..1>, J<0..1>}
18
19   group VLIW = ( {A<0..4>, LD<0..2>} . J . (A|LD) )
20 }
```

Listing 4.1: VADL Group Specification (multiple group definitions only for illustration, at most one group declaration is allowed)

$$
\begin{aligned}
\langle expr \rangle &::= \langle literal \rangle \mid \langle rep \rangle \mid \langle alt \rangle \mid \langle perm \rangle \mid \langle seq \rangle \\
&\quad \mid \text{`(`} \langle expr \rangle \text{`)`} \\
\langle literal \rangle &::= \langle identifier \rangle \\
\langle rep \rangle &::= \langle identifier \rangle \text{`<`} \langle num \rangle \text{`..`} \langle num \rangle \text{`>`} \\
\langle alt \rangle &::= \langle expr \rangle \text{`|`} \langle alt \rangle \mid \langle expr \rangle \\
\langle seq \rangle &::= \langle expr \rangle \text{`.`} \langle seq \rangle \mid \langle expr \rangle \\
\langle perm \rangle &::= \text{`\{`} \langle perm\text{-}list \rangle \text{`\}`} \\
\langle perm\text{-}list \rangle &::= \langle expr \rangle \text{`,`} \langle perm\text{-}list \rangle \mid \langle expr \rangle
\end{aligned}
$$

Listing 4.2: The extended regular expression grammar specified in BNF

Constraints

Additionally, groups matched by the regular expression have to satisfy all specified constraints, which are annotations (in square brackets) written before the group definition. There are two types of constraints: *assertions* and *stop* constraints. Assertions have to be satisfied by all constraints and are checked once after a group was successfully matched by the regular expression.

Stop constraints are checked after every instruction consumed by the regular expression to determine if matching should be *stopped*. The group has to conform to the regular expression still, but this prevents any more instructions from being consumed. This can be used to specify distributed encodings.

Listing 4.3 shows an example of some common group constraints. This architecture uses a distributed encoding with a parallel bit, meaning groups should end with an instruction with the `par` bit set to zero. The first constraints also demonstrate the first use of the group name, which can be used like an array to extract concrete instructions at a specific index (the first constraint uses the parallel bit of the last instruction in the group) and to access the group's length (used in the second constraint).

Constraints can also contain quantified expressions, as shown in listing 4.3 and listing 4.4. The bound variables are restricted to some subset of instructions which corresponds to the union of the listed operations, so the variable in `forall a in {A,LD}` will be every instruction in the group that is part of either operation A or LD (or both). Notably, an instruction that has been matched via a specific operation in the regular expression can be matched via another operation (that it is also part of) in a constraint, as demonstrated in listing 4.3 where `LD` is a superset of `LD_LOAD`. This existential (`exists`) and universal quantification (`forall`) have the same semantics as in first order logic. The only syntactical asymmetry between the two is that `exists in {...}` can be used as a shorthand for `exists newVar in {...} then true`.

```
1  instruction set architecture DSP = {
2    format F : Bits<32> =
3      { par    [21]
4      , rs2    [20..16]
5      , rs1    [15..11]
6      , rd     [10..6]
7      , opcode [5..0]
8      }
9
10   // ...
11
12   [ VLIW(VLIW.length - 1).par = 0 stop ]
13   [ VLIW.length <= 3 assert ]
14   [ forall a in {A, LD_LOAD}, b in {A, LD_LOAD} then a.rd != b.rd assert
         ]
15   group VLIW = A<0..2>.LD<0..1>.J<0..1>
16 }
```

Listing 4.3: Basic VADL Group Constraints

Notably, `forall a in {A}, b in {B} then ...`is not equivalent to `forall a in {A} forall b in {B} then ...` because for a specific `forall`, a specific instruction is never contained in multiple variables. To achieve the same behavior in the second case, `a != b & ...` has to be added to the condition.

The group indexing expressions and bound variables from quantifiers can be used in two ways: for accessing format fields (as shown in listing 4.3) and for checking membership of a specific operation (as shown at the end of listing 4.4, using the $\in$ binary operator). These instruction values are, however, also fully integrated into VADL's type (inference) system. So a format field access is only valid if that instruction expression is guaranteed to contain it (i.e. the intersection of format fields in the instructions in the specified operations), consequently group indexing expressions can only be used to access format fields that are available on all instructions (because they are not restricted so some subset as in quantified expressions).

```
VLIW.length                // number of instructions in the group
VLIW.bitLength             // number of bits in the group

forall a in {A, LD_LOAD}, b in {A, LD_LOAD} then ...
exists i in {A, J} then ...
exists in {ALU}

VLIW(1)                    // the second instruction in the group
VLIW(VLIW.length - 1)      // the last instruction in the group

VLIW(0) ∈ ALU_FP
i ∈ ALU_FP
```

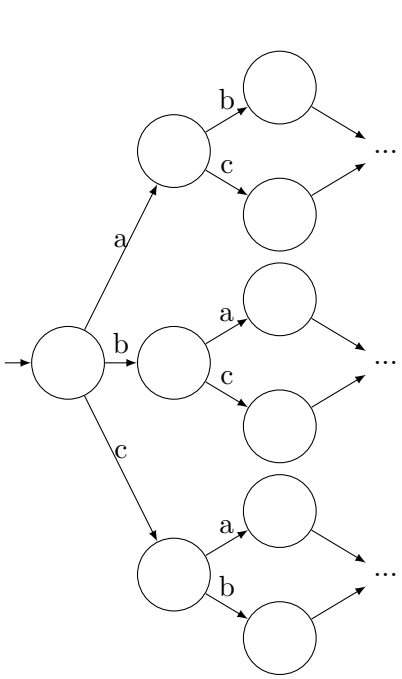
Listing 4.4: VADL Group Constraint Expression Syntax.

Group Counter

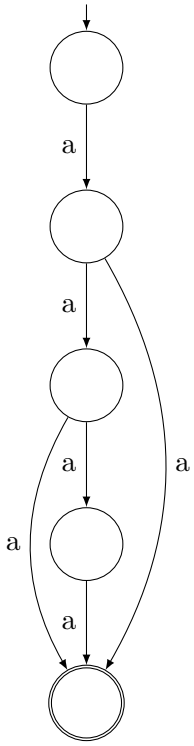
Finally, when specifying a VLIW architecture, the program counter register needs to be declared using **`group counter`** as shown in listing 4.5.

```
instruction set architecture DSP = {
  // ...

  group counter PFC : Bits<32>

  // ...

  group VLIW = /* ... */
}
```

Listing 4.5: VADL Group Program Counter



(a) The permutation {a,b,c}.



(b) The repetition a<2..4>.

Figure 4.1: Examples for a DFA representation of the regular expression extensions.

4.3 Decoding with Automatons

4.3.1 Automatons and Extended Regular Expressions

We chose regular expressions because they are easy to understand and can be efficiently parsed. But there are still multiple ways to implement extended regular expressions.

The most commonly used approach uses deterministic finite automatons (DFAs), which can express standard regular expressions very well. In our extended version however, there is the problem that permutations and repetitions can generally not easily be expressed as a DFA. (We also did not want to use nondeterministic finite automatons *NFAs*, which are generally slower and require allocating memory at runtime.)

Figures 4.1a and 4.1b illustrate the blowup in the DFA that occurs in these cases. For a permutation of n elements, all possible $n!$ orderings have to be covered by generating an automaton with 2^n states. In the case of more complex elements (not just literals), these elements have to be duplicated as well. Theoretically, the same problem also applies to repetitions, though currently only literals can be repeated and not more complex subexpressions.

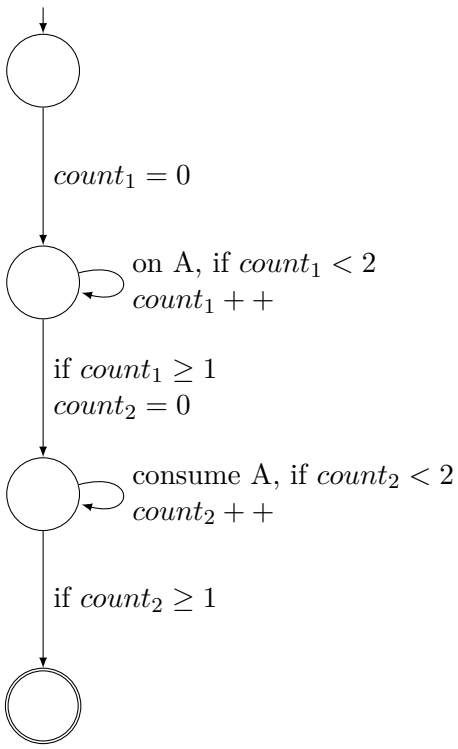


Figure 4.2: A nondeterministic counter automaton matching the regular expression A<1..2>.A<1..2>.

An alternative approach is to model these cases not with states, but with variables (called *counters*) which have no problem with enumerating these exponentially many states. Transitions between states then have *conditions* and *actions* (in addition to a symbol to consume) to read and update the counters. Somewhat counterintuitively however, a naive translation again results in a nondeterministic automaton, which is illustrated in figure 4.2 where state 1 has multiple possible next states if $count_1$ is 1. This occurs for example when parsing AAA, which could be parsed as A followed by AA or AA followed by A.

A standard regular expression can be converted into a NFA and then a DFA, e.g. using the powerset construction algorithm. But as described above, counters also introduce nondeterminism, which cannot be transformed away anymore by creating more states (without again resulting in automatons as in figure 4.1). This is because in a NFA with counters, each path might have different values for the counters. The powerset algorithm would model a NFA that might be in state X or Y as some DFA state XY, but with counters, there are multiple such states XY, each with different counter values.

Because we still wanted to implement a DFA with counters, we chose to only support a subset of extended regular expressions (which can be converted into a deterministic automaton).

This subset consists of all standard regular expressions plus all extended regular expressions where consuming a token modifies at most one counter (i.e. avoiding counter non-determinism). Some valid and invalid examples:

- A.{A,B,C}: valid
- A<1..2>.B.A<1..2>: valid
- A<1..2>.A<1..2>: invalid (reason described above)
- {AB,AC,D}: invalid (permutation has common prefix)

4.3.2 Generating Automatons

The algorithm for generating deterministic counter automatons (as described in the previous section) is not based on the commonly used Thompson's construction, but uses *pointed regular expressions* as introduced in [ACT10] (and this section reuses much of the formalism of that paper).

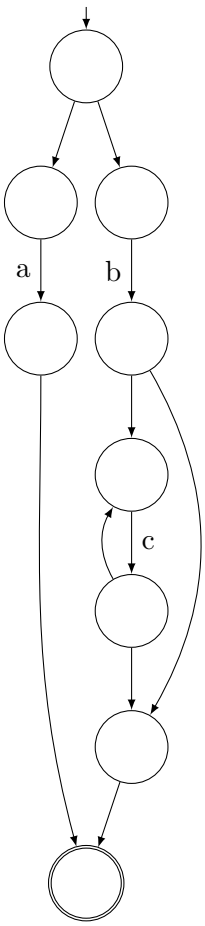
These points are inserted into a regular expression to track the intermediate states between the start and accepting states. When a token is consumed, it can be matched if and only if there exists a matching literal preceded by a point, and after that, the point moves over the token and further to the right. One way to think about this is that everything to the left of a point has already (successfully) been consumed, while everything to the right of the point still has to be consumed. Nondeterminism can be modeled by introducing multiple points, which correspond to multiple parts of the regular expression that might match the next token. This enables converting a regular expression directly into a DFA (instead of a NFA as with Thompson's construction).

Figure 4.3 demonstrates this with an example. The initial state is obtained by prefixing the regular expression with a point. In step 2, A can be matched because there exists a point followed by that token (•A) in the current state, and the point moves over (so conceptually into A•). After that, an alternative creates two points, as indeed either B or C can be consumed now. As the next token is B, the point before C is removed and only the token before B is moved along towards the right. After consuming D, the end of the alternative expression is reached and the point moves out of and after the alternative itself. This is also an accepting state as nothing remains to be parsed to the right of the point. The non-accepting trap state of NFAs corresponds to the pointed expression with no remaining points.

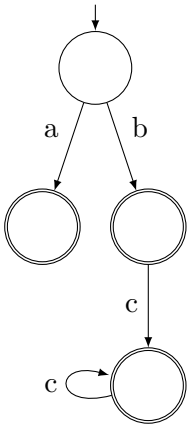
The final implementation in VADL will, however, not match strings by moving points around, but by first constructing an automaton based on this concept. The pointed regular expression are the states in that automaton and the transitions depend on which literals are preceded by points (as shown in figure 4.4c). In that specific example, the pointed automaton is minimal, that is however not always the case (see [ACT10] for a detailed discussion).

0.	Initial state	•A.(B.D|C)
1.	After consuming A	A.(•B.D|•C)
2.	After consuming B	A.(B.•D|C)
3.	After consuming D	A.(B.D|C)• (accepting state)

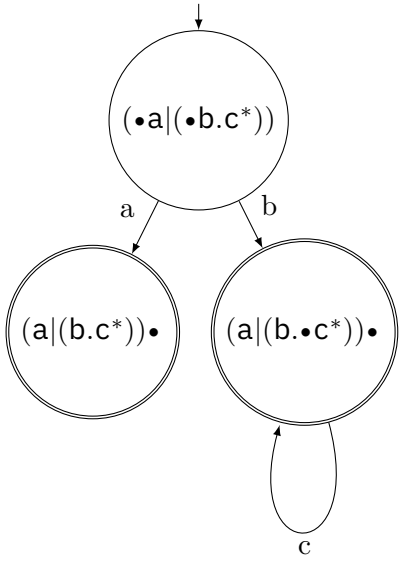
Figure 4.3: An example for using pointed regular expression to match an input string ABD against A.(B.D|C).



(a) NFA from Thompson's construction.



(b) Resulting DFA using the powerset algorithm on figure 4.4a.



(c) Directly constructed DFA using pointed expressions.

Figure 4.4: Comparing Thompson's construction with pointed expressions on the regular expression (a|(b.c*)).

Formal Specification of Extended Pointed Expressions

More formally, our extended pointed expression language (in a slightly simplified notation, $E + E$ being the alternative) conforms to the grammar

$$E ::= a \mid \bullet a \mid E + E \mid E.E \mid \{E, \ldots, E\} \mid \bullet\{E, \ldots, E\} \mid E^{a..b} \mid \bullet E^{a..b}$$

with $a \in \Sigma$ (some alphabet, such as characters or instructions). This is called a *pointed item*, a pointed expression itself is a pair $\langle e, b\rangle$ where e is a pointed item and b is a boolean signifying whether there is a point after the expression (so $e\bullet$). Notably, points are never placed in front of alternatives or sequences but always moved inside, which is not the case for permutations and repetitions as these will not be handled via duplication of pointers, but using counters.

To add a point (e.g. for the initial state or in operations introduced later on) to the start of a pointed item, the *broadcast operator* $\bullet(\cdot)$ is used. This operator is needed because points are only valid before literals, permutations and repetitions (i.e. not before alternatives and sequences). Intuitively, it moves the point inside the subexpression as necessary, e.g. $\bullet(A^{0..1} + B) = \langle \bullet A^{0..1} + \bullet B, \text{true}\rangle$ representing $(\bullet A^{0..1} + \bullet B)\bullet$.

$$\begin{aligned}
\bullet(\bullet e) &= \langle \bullet e, \text{false}\rangle \\
\bullet(a) &= \langle \bullet a, \text{false}\rangle \\
\bullet(e_1 + e_2) &= \bullet(e_1) \oplus \bullet(e_2) \\
\bullet(e_1.e_2) &= \bullet(e_1) \odot \langle e_2, \text{false}\rangle \\
\bullet(\{e_1, \ldots, e_n\}) &= \begin{cases} \langle \bullet\{e_1, \ldots, e_n\}, \text{true}\rangle & \text{if } \forall \langle e', b'\rangle \in \{\bullet(e_1), \ldots, \bullet(e_n)\}.b' = \text{true} \\ \langle \bullet\{e_1, \ldots, e_n\}, \text{false}\rangle & \text{else} \end{cases} \\
\bullet(e^{a..b}) &= \begin{cases} \langle \bullet e^{a..b}, \text{true}\rangle & \text{if } a = 0 \\ \langle \bullet e^{a..b}, \text{false}\rangle & \text{else} \end{cases}
\end{aligned}$$

where $\oplus$ and $\odot$ are helper operators:

$$\begin{aligned}
\langle e_1', b_1'\rangle \oplus \langle e_2', b_2'\rangle &= \langle e_1' + e_2', b_1' \vee b_2'\rangle \\
\langle e_1', b_1'\rangle \odot \langle e_2', b_2'\rangle &= \begin{cases} \langle e_1'.e_2', b_1'\rangle & \text{if } b_1' = \text{false} \\ \langle e_1'.e_2'', b_2' \vee b_2''\rangle & \text{else, with } \bullet(e_2') = \langle e_2'', b_2''\rangle \end{cases}.
\end{aligned}$$

For literals, permutations and repetitions, the point is added before the corresponding expression. Furthermore, permutations and repetitions can in some cases match zero length inputs (so a trailing point has to be added). For alternatives, the point is added to both subexpressions (and there is a point after the expression if at least one of the subexpressions ends with a point). For sequences, either the first subexpression is still being matched, or the first subexpression is finished and thus the point has to be moved into the second subexpression.

The broadcast operator is then also extended to pointed expressions:

$$\bullet(\langle e, b\rangle) = \langle \bullet e', b \vee b'\rangle \text{ where } \bullet(e) = \langle e', b'\rangle$$

The central operation of pointed expressions is the *move* operation, which consumes the next input token and moves the points accordingly (as shown in figure 4.3).

This is also where conditions and actions are first introduced. They are specified via a counter identifier. There are two different action types:

- ActBitSet(c, idx: $\cdot$) which sets the idx-th bit of counter named c, and
- ActInc(c) which increments the counter c by one.

The possible condition types are:

- CondGrEq(c, v: $\cdot$) to check if the counter value is greater or equal the given value,
- CondLe(c, v: $\cdot$) to check if the counter value is smaller than the given value,
- CondBitCleared(c, idx: $\cdot$) to check if the idx-th bit is set in the counter value,
- CondRestFull(c, exc: es, width: w) to check if the counter has the lowest w bits set (except for the bits in the list es which can be unset still), and
- CondRestRemains(c, exc: es, width: w) to check if the counter has an unset bit (which is not in the list es) in the lowest w bits.

$move(e, t, p)$ gives a list of transitions when reading a token t in state (i.e. pointed expression) e, each with a next state (pointed expression), conditions and actions. p is a string that is used to identify the current position in the expression to use as the counter identifier. Its value is arbitrary but has to be unique for every element of the expression.

First, for the unpointed case, literal expressions don't change (in equation (4.1)), and sequences and permutations use a version of the helper operators that also collect any conditions and actions from the recursive calls (in equations (4.2) and (4.3)).

For repetitions, there are two possibilities (in equation (4.4)): if there's no trailing point in the processed subexpression (b' is false) then nothing should happen as any potential points are still moving inside e', otherwise the current iteration of the repetition is finished and there are two possible next states (either another iteration is needed because the counter is below the minimum value and so the point is moved back to the start, or the whole repetition was successfully matched and the true in the tuple denotes a trailing point). Note how this can introduce nondeterminism as both of these conditions might be true.

For permutations (equation (4.5)), all possible combinations of the processed subexpressions have to be considered. If none of the subexpressions have a trailing point ($|I'| = 0$), then once again nothing should happen while the points move inside of the expression.

$$move(a, t, p) = \{\langle a, \text{false}, \{\}, \{\}\rangle\} \tag{4.1}$$

$$move(e_1 + e_2, t, p) = move(e_1, t, \text{"}p.0\text{"}) \oplus_c move(e_2, t, \text{"}p.1\text{"}) \tag{4.2}$$

$$move(e_1.e_2, t, p) = move(e_1, t, \text{"}p.0\text{"}) \odot_c move(e_2, t, \text{"}p.1\text{"}) \tag{4.3}$$

$$move(e^{a..b}, t, p) = \tag{4.4}$$

$$\bigcup_{\langle e', b', C', A'\rangle \in E'} \begin{cases} \{\langle e'^{a..b}, \text{false}, C', A'\rangle\} & \text{if } b' = \text{false} \\ \{\ \langle \bullet e'^{a..b}, \text{false}, C' \cup \{\textsc{CondLe}(p, \text{v: } a)\}, A' \cup \{\textsc{ActInc}(p)\}\rangle, \\ \quad \langle e'^{a..b}, \text{true}, C' \cup \{\textsc{CondGrEq}(p, \text{v: } a)\}, A' \cup \{\textsc{ActInc}(p)\}\rangle \\ \} & \text{else} \end{cases}$$

$$\text{with } E' = move(e, t, \text{"}p.0\text{"})$$

$$move(\{e_1, \ldots, e_n\}, t, p) = \tag{4.5}$$

$$\bigcup_{\substack{\langle e'_1, b'_1, C'_1, A'_1\rangle \in E'_1, \ldots, \\ \langle e'_n, b'_n, C'_n, A'_n\rangle \in E'_n}} \begin{cases} \{\langle \{e'_1, \ldots, e'_n\}, \text{false}, C', A'\rangle\} & \text{if } |I'| = 0 \\ \{\langle \bullet\{e'_1, \ldots, e'_n\}, \text{false}, C' \cup \{\textsc{CondRestRemains}(p, \text{exc: } I', \text{width: } n), \\ \quad \textsc{CondRestRemains}(p, \text{exc: } I' \cup I'_\epsilon, \text{width: } n)\}, \bar{A}'\rangle, \\ \ \langle \bullet\{e'_1, \ldots, e'_n\}, \text{true}, C' \cup \{\textsc{CondRestRemains}(p, \text{exc: } I', \text{width: } n), \\ \quad \textsc{CondRestFull}(p, \text{exc: } I' \cup I'_\epsilon, \text{width: } n)\}, \bar{A}'\rangle, \\ \ \langle \{e'_1, \ldots, e'_n\}, \text{true}, C' \cup \{\textsc{CondRestFull}(p, \text{exc: } I', \text{width: } n)\}, \bar{A}'\rangle \\ \} & \text{else} \end{cases}$$

$$\text{with } C' = \bigcup_{1 \le i \le n} C'_i \text{ and } A' = \bigcup_{1 \le i \le n} A'_i$$

$$\text{with } \bar{A}' = A' \cup \{\textsc{ActBitSet}(p, \text{idx: } i) \mid i \in I'\}$$

$$\text{with } I' = \{i \in \mathbb{N} \mid 1 \le i \le n, b'_i = \text{true}\}$$

$$\text{with } I'_\epsilon = \{i \in \mathbb{N} \mid 1 \le i \le n, \bullet(e_i) = (\cdot, \text{true})\}$$

$$\text{with } E'_i = move(e_i, t, \text{"}p.i\text{"})$$

Otherwise, there are three possibilities: if there are still unset bits remaining in the counter (even when excluding the bits corresponding to subexpressions that match a zero-length input), then a point is added to the beginning again. Secondly, if there are still unmatched subexpressions but all of them match zero-length input, then a point is added to the beginning to potentially keep matching those, but the expression also gets a trailing point. Finally, if every single subexpression was matched, the permutation cannot be visited again and there is only a trailing point.

The remaining cases are now the pointed expressions. A point is allowed to move over a literal if it is the same symbol that is currently being consumed (equation (4.8)), otherwise the point is discarded.

For pointed permutations (equation (4.9)), we need to ensure that points already inside

$$\{\langle e_1, b_1, cs_1, as_1\rangle, ..., \langle e_m, b_m, cs_m, as_m\rangle\} \oplus_c \{\langle e'_1, b'_1, cs'_1, as'_1\rangle, ..., \langle e'_n, b'_n, cs'_n, as'_n\rangle\} = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \langle e_i \oplus e'_j, cs_i \cup cs'_j, as_i \cup as'_j\rangle \tag{4.6}$$

$$\{\langle e_1, b_1, cs_1, as_1\rangle, ..., \langle e_m, b_m, cs_m, as_m\rangle\} \odot_c \{\langle e'_1, b'_1, cs'_1, as'_1\rangle, ..., \langle e'_n, b'_n, cs'_n, as'_n\rangle\} = \bigcup_{i=1}^{m} \bigcup_{j=1}^{n} \langle e_i \odot e'_j, cs_i \cup cs'_j, as_i \cup as'_j\rangle \tag{4.7}$$

continue to be processed, but additionally, the point can move into exactly one of the n subexpressions of the permutation. For each of these, there are three cases (similar to the non-pointed permutation case described above): either the permutation still isn't matched yet and has to be revisited, the permutation has been matched but there are still optional subexpressions, or the permutation is fully matched and cannot be visited again.

For repetitions (equation (4.10)), if the subexpression isn't already matched after a single token, the point moves inside the subexpression. If it is matched after one token, then there are again three possibilities: the minimum amount wasn't yet reached so it should be revisited, or the counter is between the minimum and maximum amount, so the repetition was successfully matched but can still be revisited, or finally the maximum number was reached and the point is now placed only after the repetition.

$$move(\bullet a, t, p) = \begin{cases} \{\langle a, \text{true}, \{\}, \{\}\rangle\} & \text{if } a = t \\ \{\langle a, \text{false}, \{\}, \{\}\rangle\} & \text{else} \end{cases} \tag{4.8}$$

$$move(\bullet\{e_1, \ldots, e_n\}, t, p) = \tag{4.9}$$

$$\begin{cases} move(\{e_1, \ldots, e_n\}, t, p) & \text{if } \{e_1, \ldots, e_n\} \text{ contains a point} \\ \{\} & \text{else} \end{cases}$$

$$\bigcup$$

$$\bigcup_{i=1..n} \bigcup_{(e'_i, b'_i, C'_i, A'_i) \in E'_i} \begin{cases} \{\langle \bullet\bar{e}'_i, \text{false}, C' \cup \{\textsc{CondBitCleared}(p, \text{idx: } i), \\ \quad \textsc{CondRestRemains}(p, \text{exc: } \{i\}, \text{width: } n) \\ \quad \textsc{CondRestRemains}(p, \text{exc: } I'_\epsilon \cup \{i\}, \text{width: } n)\}, \bar{A}'_i\rangle, \\ \langle \bullet\bar{e}'_i, \text{true}, C' \cup \{\textsc{CondBitCleared}(p, \text{idx: } i), \\ \quad \textsc{CondRestRemains}(p, \text{exc: } \{i\}, \text{width: } n) \\ \quad \textsc{CondRestFull}(p, \text{exc: } I'_\epsilon \cup \{i\}, \text{width: } n)\}, \bar{A}'_i\rangle, \\ \langle \bar{e}'_i, \text{true}, C' \cup \{\textsc{CondBitCleared}(p, \text{idx: } i), \\ \quad \textsc{CondRestFull}(p, \text{exc: } \{i\}, \text{width: } n)\}, \bar{A}'_i\rangle \\ \} & \text{if } b'_i = \text{true} \\ \{\langle \bar{e}'_i, \text{false}, C'_i, A'_i\rangle\} & \text{if } e'_i \text{ contains a point} \\ \{\} & \text{else} \end{cases}$$

$$\text{with } \bar{A}'_i = A'_i \cup \{\textsc{ActBitSet}(p, \text{idx: } i)\}$$
$$\text{with } \bar{e}'_i = \{e_1, \ldots, e_{i-1}, e'_i, e_{i+1}, \ldots, e_n\}$$
$$\text{with } E'_i = move(\bullet(e_i), t, \text{"}p.i\text{"})$$
$$\text{with } I'_\epsilon = \{i \in \mathbb{N} \mid 1 \le i \le n, \bullet(e_i) = (\cdot, \text{true})\}$$

$$move(\bullet e^{a..b}, t, p) = \tag{4.10}$$

$$\bigcup_{\langle e', b', C', A'\rangle \in E'} \begin{cases} \{ \\ \quad \langle e'^{a..b}, \text{false}, C' \cup \begin{cases} \{\textsc{CondLe}(p, \text{v: } b-1)\} & \text{if } e' \text{ has a point} \\ \{\} & \text{else} \end{cases}, A'\rangle \\ \} & \text{if } b' = \text{false} \\ \{ \\ \quad \langle \bullet e'^{a..b}, \text{false}, \\ \qquad C' \cup \{\textsc{CondLe}(p, \text{v: } a-1)\}, \\ \qquad A' \cup \{\textsc{ActInc}(p)\}\rangle, & \text{if } a > 1 \\ \quad \langle \bullet e'^{a..b}, \text{true}, \\ \qquad C' \cup \{\textsc{CondGrEq}(p, \text{v: } a), \textsc{CondLe}(p, \text{v: } b-1)\}, \\ \qquad A' \cup \{\textsc{ActInc}(p)\}\rangle, & \text{if } b > 1 \\ \quad \langle e'^{a..b}, \text{true}, \\ \qquad C' \cup \{\textsc{CondGrEq}(p, \text{v: } b)\}, \\ \qquad A' \cup \{\textsc{ActInc}(p)\}\rangle \\ \} & \text{else} \end{cases}$$

$$\text{with } E' = move(\bullet(e), t, \text{"}p.0\text{"})$$

4.4 Constraints

For the simulator, each constraint (with the syntax that was introduced in section 4.2) is compiled into a C++ function. These functions receive an environment containing the bundle matched by the automaton from section 4.3, which is used by the generated code to retrieve the concrete instructions for e.g. `VLIW(i)` and the quantified expressions such as **`forall`**.

The quantified expressions iterate over the whole group and execute the condition for each matching instruction, the remaining expression syntax is compiled in the same way as done in other VADL constructs, such as instructions.

The abstract instruction values (which appear like objects in the syntax, such as `VLIW(VLIW.length-1).par != 0`) are tuples in C++, containing the instruction word (`uint64_t`), the instruction type (**`enum`** `CPU_Instructions`), and the index inside the bundle (`uint32_t`). This way,

- `a.rd` calls a field getter function with the instruction word: `extract_rd(std::get<0>(a))`,
- `a ∈ ALU_FP` uses the instruction type and checks it against the set representing the operation: `SET_ALU_FP.includes(std::get<1>(a))`, and
- `a != b` stays `a != b`. Multiple identical instructions inside a bundle still satisfy `a != b` because while the instruction word and type are identical, the index still differs.

Stop constraints are translated in the same way as assertions, they are merely executed after every transition of the automaton and prevent the automaton from greedily matching as many instructions as possible.

4.5 Simulation

The previous two sections cover the decoding of a VLIW bundle, but the simulator still has to correctly execute the instructions in the bundle.

The difference to non-VLIW architectures is that all instructions in a single bundle execute in parallel, meaning that writes to registers should only be visible in the next bundle. For example in a bundle containing the two instructions {R0 = R0 + 1, R1 = R0 - 2}, the subtraction should use the old, non-incremented value of R0.

The simulator achieves this by executing each instruction twice, to first read all operands (without writing back any results), and then again to use the buffered operands to compute the result and write back the results. This is illustrated for a simple addition instruction in figure 4.5. The buffered operand is stored in a slot based on the index inside the bundle i to prevent multiple instructions of the same type in a single bundle

from overwriting each other because the buffer is essentially a static variable that exists once only per instruction.

Algorithm 4.1 contains pseudocode for everything that happens in a single loop of the simulator. First, a chunk of memory is fetched which is as long as the longest possible bundle. Then the bundle is parsed with the automaton, which determines the individual instructions inside the bundle and its length (number of instructions) and size (in bytes). The assertions are checked against the bundle, causing the simulator to stop if there's a violated assertion. Then the two phases of instruction execution are performed. Finally, the program counter is incremented to point to the next bundle in case no jump instruction was executed already.

Algorithm 4.1: A single step of the simulation loop

Input: The bundle counter pc and the maximum bundle size $maxsize$.
Output: Executes the next bundle.

$fetchWord \leftarrow$ memory$[pc \,..\, (pc + maxsize)]$;
$(bundle, bundleSize) \leftarrow$ executeAutomaton$(fetchWord)$;
if *!*assertionsSatisfied*(bundle)* **then**
  throw;
end
$pc' \leftarrow pc$;
$pc.currentSize \leftarrow bundleSize$;
for $i = 0$ **to** $bundle.length - 1$ **do**
  executeInstruction$(bundle[i], i,$ writeback: false$)$;
end
for $i = bundle.length - 1$ **to** 0 **do**
  executeInstruction$(bundle[i], i,$ writeback: true$)$;
end
if $pc' = pc$ **then**
  $pc \leftarrow pc + bundleSize$;
end

The second for-loop is inverted because, for most architectures, the order does not matter (as they should be executed in parallel), while for example Hexagon has some cases where this reversed ordering is preferable (see section 5.1.2).

VADL has two functionalities to make working with the program counter easier: `[next] program counter PC` (meaning that `PC` doesn't point to the current instruction but the next instruction) and `PC.current`/`PC.next` (to retrieve the next instruction address even regardless of whether the program counter has `[next]`), which can be handled easily for non-VLIW architectures because in these cases, `PC.next - PC.current` is a constant (the size of the current (e.g. jump) instruction, so usually 32 or 64 bit). But for VLIW architectures, this is actually the size of the bundle, which is therefore stored in a variable to compute the value of `PC.current`/`PC.next` expressions, when used.

(a) The non-VLIW behavior.

Input: Format fields $rs1, rs2, rd$.

Input: Register file X.

Output: Executes the instruction.

$a \leftarrow X.read(rs1);$

$b \leftarrow X.read(rs2);$

$result \leftarrow a + b;$

$X.write(rd, result);$

(b) The two-phased VLIW behavior.

Input: Format fields $rs1, rs2, rd$.

Input: Register file X.

Input: *writeback* flag, index i.

Output: Executes the instruction.

if *writeback* **then**
  $a \leftarrow aBuf[i];$
else
  $a \leftarrow X.read(rs1);$
  $aBuf[i] \leftarrow a;$
end

if *writeback* **then**
  $b \leftarrow bBuf[i];$
else
  $b \leftarrow X.read(rs2);$
  $bBuf[i] \leftarrow b;$
end

$result \leftarrow a + b;$

if *writeback* **then**
  $X.write(rd, result);$
end

Figure 4.5: The translated instruction behavior of an add instruction performing $X(rd) := X(rs1) + X(rs2)$, for non-VLIW and VLIW architectures.

CHAPTER 5

Evaluation

We evaluated the implemented VLIW functionalities along two axes: whether it is possible to implement two existing VLIW architectures in VADL (qualitative), and the performance of the functional VLIW instruction set simulator (quantitative).

5.1 Case Studies of Two Architectures

We implemented a subset of two architectures to determine how well VLIW architectures can be modeled with the changes to VADL. In both cases, only some instructions were implemented (because there are simply too many of them), and a few features of the architectures themselves posed problems.

5.1.1 Texas Instruments C6x

The Texas Instruments C6x architecture has very permissive bundle grouping and ordering restrictions: each instruction belongs to one of the eight functional units, and each bundle may contain at most one instruction for each unit, in any order. Listing 5.1 shows how this is modeled in VADL using a permutation containing optional elements for each of the units (which are defined as operations). The stop constraint uses the parallel bit to stop the decoder early as already described. This behaves correctly even for the valid case where a bundle contains no such stop bit, because the stop constraint does not need to be satisfied for a bundle to be valid. The decoder simply stops after at most eight instructions (which is the length of the permutation).

```
1 [VLIW(VLIW.length - 1).par = 0 stop]
2 group VLIW = ({
3     L1<0..1>, L2<0..1>, S1<0..1>, S2<0..1>,
4     D1<0..1>, D2<0..1>, M1<0..1>, M2<0..1> })
```

Listing 5.1: Group Expression for C6x

The dual data paths with separate register files and the register file cross path can be specified as separate instructions, which is simplified greatly by using macros (just like addressing modes are specified using separate instructions for each mode). For example, there are separate VADL instructions for the single (architectural) addition instruction: `ADD_L1dA`, `ADD_L1dB`, `ADD_L2dA`, and `ADD_L2dB` (L1 and L2 are the units, and A/B are the two cross path versions), plus 12 additional versions each for all the possible predications.

Relative jumps are not relative to the jump instruction itself (like in non-VLIW architectures), but to the start of the fetch bundle, which turns out to be the same as the 8-word aligned address of the current execute bundle. Listing 5.2 shows how the relative `CALLP` instruction might be implemented.

```
1 // The start of the current fetch package (256 bit aligned)
2 function PCE1 -> Address = (PFC >> 8) << 8
3
4 [operation S1]
5 instruction CALLP_S1 : BranchDispFormat = {
6     A(3) := PFC.next // return address
7     PFC := PCE1 + ((cst21, 0b00 as Bits<2>) as SInt)
8 }
```

Listing 5.2: Calculating relative branch targets on C6x

C6x has exposed instruction latencies in the ISA via delay slots for almost all instructions (meaning not just the usual branch delay slot). There are plans for VADL to support this (see chapter 6), but for now, the implementation has effectively no latencies for all instructions. Consequently, this means that all programs generated by standard compilers for this architecture cannot run in the VADL simulator due to differing instruction latencies.

Furthermore, this also complicates the constraints because for example register or memory writes occur n cycles after the corresponding instruction was dispatched in a bundle. So static constraints for each bundle are not enough and dynamic constraints over sequences of bundles are necessary (once again, see chapter 6).

Some newer variants of C6x also support *compact instructions* which allow compressing a subset of instructions into 16-bit instruction words instead of the usual 32-bit words. VADL currently doesn't support this (see chapter 6 for a potential solution).

5.1.2 Qualcomm Hexagon

The bundles in Qualcomm's Hexagon architecture have a strict ordering, which was straight forward to specify in the group declaration (see listing 5.3). Each of the ten

instruction classes is defined as a VADL operation, though there are some `SYSTEM` instructions which cannot be part of a bundle, which are listed in the separate operation `SYSTEM_SOLO`. A bundle end is marked with the 2-bit `parse` field containing all ones or all zeros, the other two values are used for hardware loops, which currently cannot be implemented in VADL. The regular expression only allows at most four instructions in a bundle, but as explained in section 2.1, bundle words can be at most four 32-bit words long (due to constant extenders), which is specified in the second constraint in listing 5.3.

```
[ let parse = VLIW(VLIW.length - 1).parse in ((parse=0b00) | (parse=0b11
    )) stop ]
[ VLIW.bitLength <= ((4*32) as UInt<32>) assert ]
group VLIW = (
  (
   (XTYPE<0..1> | ALU32<0..1> | J<0..1> | CR<0..1>).
   (XTYPE<0..1> | ALU32<0..1> | J<0..1> | JR<0..1> | CR23<0..1>).
   (LD<0..1> | ST<0..1> | ALU32<0..1>).
   (LD<0..1> | ST<0..1> | ALU32<0..1> | MEMOP<0..1> | NV<0..1> | SYSTEM
       <0..1>)
  )
  | SYSTEM_SOLO
)
```

Listing 5.3: Group Expression and Base Constraints for Hexagon

Due to constant extenders, all eligible instructions need to be specified twice, with and without constant extenders. Listing 5.4 demonstrates this for a single immediate-to-register-transfer instruction. In the actual specification, the last few lines can be specified more concisely using macros, but for now the two instructions formats have to be specified manually still.

```
function extendSExtender(imm: SInt<32>, extender: Bits<26>) -> SInt<32>
    =
    (extender, imm as Bits<6>)

format TransferImmFormat : Bits<32> = {
    iclass [31..28]
  , rs     [27]
  , majop  [26..24]
  , parse  [15..14]
  , imm    [23..22, 20..16, 13..5]
  , d5     [4..0]
  , immS   = (imm as SInt) as SInt<32> }
format TransferImmXFormat : Bits<64> = {
    iclassX [31..28], parseX [15..14], immX [27..16,13..0]
  // ---
  , iclass  [63..60]
  , rs      [59]
  , majop   [58..56]
```

```
  , parse   [47..46]
  , imm     [55..54, 52..48, 45..37]
  , d5      [36..32]
  , immS    = (immX, imm as Bits<6>) }

[ operation ALU32 ]
instruction TransferImm  : TransferImmFormat = { R(d5) := immS  }
encoding TransferImm = { iclass=0b0111, rs=0b1, majop=0b000 }
[ operation ALU32 ]
instruction TransferImmX : TransferImmXFormat = { R(d5) := immS }
encoding TransferImmX= { iclass=0b0111, rs=0b1, majop=0b000, iclassX=0 }
```

Listing 5.4: Hexagon Register-Immediate Transfer Instruction with Constant Extenders

Listing 5.5 defines the operations of the instruction classes already used above, and more operations which will be needed to specify the remaining grouping and ordering constraints.

```
// The instruction classes which form a partition over all instructions
operation XTYPE = { XTYPE_PRED_WRITING }
operation ALU32 = { ALU32_PRED_WRITING }
operation CR = { CR23 }
operation JR = {}
operation J = { J_PRED_WRITING }
operation LD = {}
operation MEMOP = {}
operation NV = {}
operation ST = {}
operation SYSTEM = {}
operation SYSTEM_SOLO = {}

operation ALL = { XTYPE, ALU32, CR, JR, J, LD, MEMOP, NV, ST, SYSTEM,
    SYSTEM_SOLO }

// may execute on either slot2 or slot3, even though it is CR-type
operation CR23 = {}

operation TRANSFER_C_R = { TransferCR }
operation TRANSFER_CC_RR = { TransferCCRR }
operation TRANSFER_P_R = { TransferPR }
operation ARITH_DOUBLE_CARRY = { AddDoubleCarry, SubDoubleCarry }
operation MEMLOCKED_L2FETCH_TRACE = { memw_locked, memd_locked, l2fetch,
     trace }
operation CACHE_MAINTENANCE_SPECIAL = { dccleana, dcinva, dccleaninva,
    dczeroa }
operation JUMP_IMM = { JumpImm }
operation JUMP_CMP_NEW = { } //  if ([!]cmp.xx(Rs.new, Rt)) jump

operation ALU32_XTYPE_NON_FP = { ALU32, XTYPE_NONFP }
operation CONDITIONAL_NEW = {} // conditional instrs with a dot-new
    predicate
```

```
30 operation FLOATING_POINT = {}
31 operation ARITH_SATURATING = { AddRegSat, SubRegSat }
32 operation DEALLOC_RETURN = { DeallocReturn, DeallocReturnPredPos,
       DeallocReturnPredNeg }
33 operation LD_ST = { LD, ST }
34
35 operation ALU32_PRED_WRITING = {  } // helper operation
36 operation J_PRED_WRITING = {  }     // helper operation
37 operation XTYPE_PRED_WRITING = {  } // helper operation
38 operation PRED_WRITING = { ALU32_PRED_WRITING, J_PRED_WRITING,
       XTYPE_PRED_WRITING } // all predicate writing instructions except
       transfers
39
40 function implies(a: Bool, b: Bool) -> Bool = (!a) | b
```

Listing 5.5: Hexagon Operation Sets

Using these operations, we specified all constraints of the Hexagon ISA (excluding the ones related to hardware loops), which are listed below. There were no major problems, however some of these constraints could be specified more easily if operations could be filtered using constraints as well (e.g. `PRED_WRITING` would also contain those register transfer instructions where the destination register is equal to 4).

- Dot-new conditional instructions need an instruction generating the predicate

  ```
  [forall i in {CONDITIONAL_NEW} then ((i.dn = 0) | (exists j in {
      PRED_WRITING} then j.d2 = i.u2)) assert ]
  ```

- May not be first in dual jump: unconditional `jump`

  ```
  [!( (VLIW.length >= 2) & ((VLIW(0) ∈ JUMP_IMM) & (VLIW(1) ∈ J)) )
      assert ]
  ```

- May not be in dual jump: `if ([!]cmp.xx(Rs.new, Rt))jump`

  ```
  [!( (VLIW.length >= 2) & (((VLIW(0) ∈ JUMP_CMP_NEW) & (VLIW(1) ∈ J
      )) | ((VLIW(0) ∈ J) & (VLIW(1) ∈ JUMP_CMP_NEW))) ) assert ]
  ```

- May not be in dual jump: all `jumpr` (JR is only allowed in slot 1)

  ```
  [!( (VLIW.length >= 2) & ((VLIW(0) ∈ J) & (VLIW(1) ∈ JR))) assert]
  ```

- May not be in dual jump: all `dealloc_return`

  ```
  [!( (VLIW.length >= 2) & (((VLIW(0) ∈ J) & (VLIW(1) ∈ J)) & exists
       in {DEALLOC_RETURN}) ) assert ]
  ```

- `memw_locked`, `memd_locked`, `l2fetch`, and `trace` must execute on Slot 0. They must be grouped only with ALU32 or (non-FP) XTYPE instructions.

```
[implies(VLIW(VLIW.length-1) ∈ MEMLOCKED_L2FETCH_TRACE, forall j
    in {ALL} then ((j = VLIW(VLIW.length-1)) | (j ∈
    ALU32_XTYPE_NON_FP)) ) assert ]
```

- `dccleana`, `dcinva`, `dccleaninva`, and `dczeroa` must execute on Slot 0. Slot 1 must be empty or an ALU32 instruction. Note that Slot 1 being empty means that Slot 2 or 3 follow immediately.

```
[implies(VLIW(VLIW.length-1) ∈ CACHE_MAINTENANCE_SPECIAL, (VLIW.
    length = 1) | !(VLIW(VLIW.length-2) ∈ LD_ST)) assert ]
```

- Instructions in a packet cannot unconditionally write to the same destination register.
  This is not fully specified yet. However, once the `[operation ...]` annotation syntax can be used to easily add an instruction to multiple operation sets, every instruction with a destination register would be added to some `operation REG_WRITING`. Every such instruction must also have a format field access function such as `d5` (to have a uniform variable to reference the target register). Then, the constraint would look like this, ensuring that the destination register is different for all pairs of register writing instructions:

```
[forall i,j in {REG_WRITING} then (i.d5 != j.d5) assert ]
```

- Instructions that write to the entire user status register are not allowed to be grouped in a packet with any instruction that writes to a bit in the user status register.

```
[!( (exists i in {TRANSFER_C_R,TRANSFER_CC_RR} then i.d5 = 8) &
    exists in {FLOATING_POINT, ARITH_SATURATING} ) assert ]
```

- If a packet contains a register transfer from a general register to a predicate register, then no other instruction in the packet can write to the same predicate register.

```
[!( ((exists i in {TRANSFER_C_R,TRANSFER_CC_RR} then i.d5 = 4) | (
    exists in {TRANSFER_P_R})) &
        exists in {PRED_WRITING} ) assert ]
```

- The instructions `spNloop0`, `decbin`, `tlbmatch`, `memw_locked`, `memd_locked`, `add:carry`, `sub:carry`, `sfcmp`, and `dfcmp` cannot be grouped with another instruction that sets the same predicate register.

```
[forall i in {ARITH_DOUBLE_CARRY, CMP_FP} then !(
    ((exists j in {PRED_WRITING} then (i != j) & (j.d2 = i.u2)) |
     (exists j in {TRANSFER_C_R,TRANSFER_CC_RR} then j.d5 = 4)) |
     (exists j in {TRANSFER_P_R} then j.d2 = i.u2 )
) assert ]
```

Dual jump and store instructions work automatically because instructions in a bundle

are executed in reverse order (see section 4.5).

Hexagon has an ordinary general register and a control register file (see listing 5.6), the program counter is architecturally stored in the control register file, but VADL currently doesn't fully support using a register file entry as the group counter via `alias group counter PC: Bits<32> = C(9)` (which required some workarounds in instructions that might access `C(9)`). As already alluded to in section 4.5, Hexagon's program counter points to the next bundle as opposed to the current bundle's address, requiring a `[next]` annotation.

The four predicate registers are stored in a single register with four subregisters (specified by the `partial` annotations, this is however the default value anyway). This enables multiple predicates to be written to in the same bundle. Without subregisters, the instructions would read the old value of all four predicates and write back the updated value, resulting in only one predicate to be updated correctly (at least in the simulator backend).

```
1  register file R   : Bits<5> -> Bits<32> // general purpose registers
2  register file C   : Bits<5> -> Bits<32> // control registers
3
4  alias register USR : USRFormat = C(8)   // User status register
5  alias register GP              = C(11)  // Global pointer register
6  // (remaining registers omitted)...
7
8  [next]
9  group counter PC : Bits<32>
10
11 format PFormat : Bits<32> =
12   { P3   : Bits<8>
13   , P2   : Bits<8>
14   , P1   : Bits<8>
15   , P0   : Bits<8>
16   }
17 [ read partial ]
18 [ write partial ]
19 register P : PFormat
```

Listing 5.6: Hexagon Registers

New-value instructions can currently not be specified due to their interaction of instructions inside a bundle. Automatic AND-ing of parallel predicate register writes doesn't work for the same reason.

Duplex instructions are currently not specified, there is however a plan to be able to specify them via **protocol**s (see chapter 6 for more details).

As already mentioned, hardware loops are not supported as they require the bundle itself to have a behavior (as opposed to only instructions driving the behavior), but this could potentially also be solved using **protocol**s (again, see chapter 6).

5.2 Simulator Performance

Finally, the performance of the VLIW-capable ISS was evaluated using an VADL specification of a subset of the Hexagon ISA implementing 425 instructions.

LLVM models all control flow (i.e. if branches and for/while loops) with bundles containing new-value instructions on predicate registers (such as `{P0 = cmp.eq(R2,#4); if (P0.new)jump:t exit }`). It cannot be configured to instead generate two separate bundles for these two instructions. New-value instructions can currently not be specified in VADL, so for all benchmarks, the LLVM output was post-processed to split up such bundles into a sequence of two bundles (such as `{P0 = cmp.eq(R2,#4)} {if (P0)jump:t exit }`).

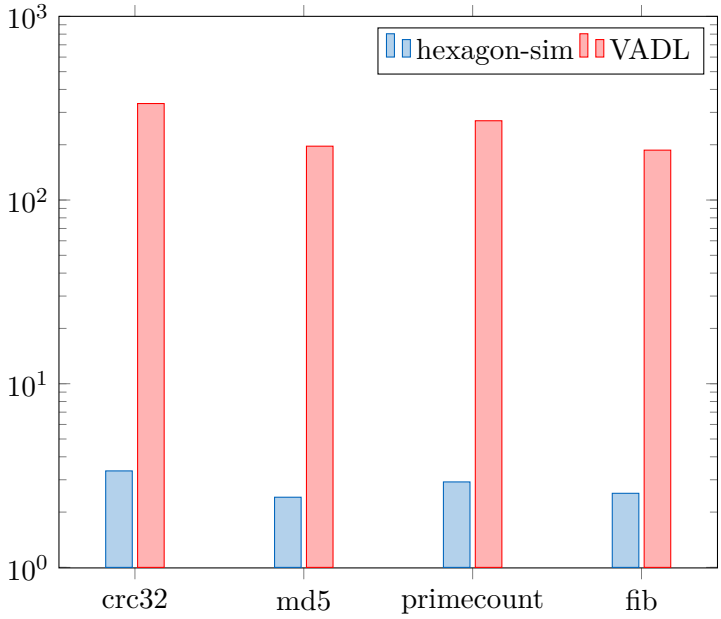
In this evaluation, the VADL ISS was compared against QEMU (which uses dynamic binary translation), and Qualcomm's own (proprietary) Hexagon instruction set simulator `hexagon-sim` (which appears to be an interpreter, though there is no official documentation clearly stating this). To be able to run the same binary in all simulators, a QEMU version based on v9.2.50 but supporting Hexagon system emulation, which is currently being developed by Qualcomm (and not yet upstreamed into QEMU)[1], was used.

The benchmarks used were a subset of programs from the Embench[2] benchmark suite, and a branchless recursive implementation of the Fibonacci sequence computing the 16th Fibonacci number. Figure 5.1a shows the relative runtime of QEMU, `hexagon-sim` and the VADL ISS. Relative to QEMU, `hexagon-sim` is 3 times slower while the VADL ISS is 250 times slower.

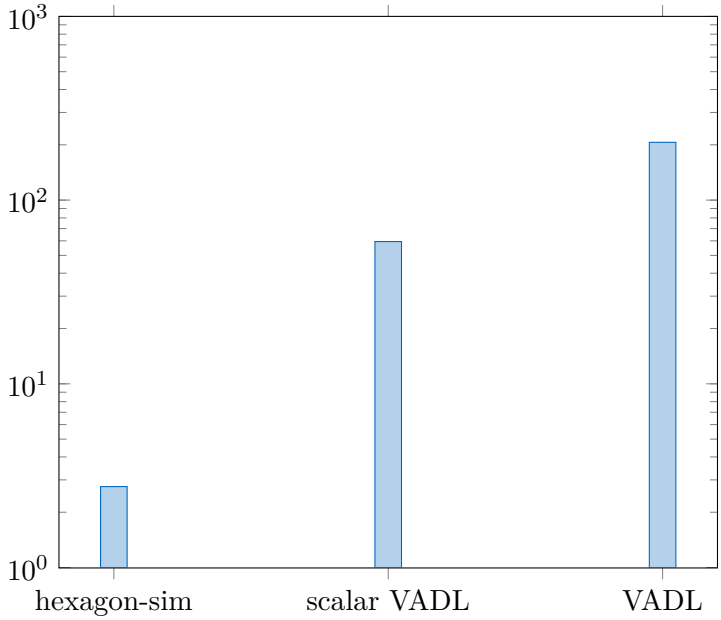
The VADL ISS is much slower than the other simulators (which was already a known issue and has lead to multiple new iterations of implementations using direct-threaded code and most recently generating a QEMU backend instead). See chapter 6 for a more comprehensive overview of VADL simulator performance improvements. While implementing the Hexagon specification, we also noticed that the simulator became significantly slower as more instructions were added (around 5 times slower when going from a 240 instruction specification to the final version with 425 instructions), as VADL's generated decoder has to perform more comparisons.

To quantify any VLIW specific overhead, the same VADL specification was used to generate an ISS that implements the same instruction set but in a scalar version, i.e. without the **`group`** and **`group counter`** definitions. Together with a benchmark binary that was compiled to contain only bundles of length one, this scalar ISS can be compared against the VLIW ISS (see Figure 5.1b), showing that the VLIW specific simulator aspects (the group parsing, and two-pass instruction execution) lead to a 3.4x slowdown. This can be explained by the VLIW version having to execute the decoder multiple times (not strictly necessarily, but to work with the ISS's existing implementation).

[1] `https://github.com/quic/qemu/pull/99`
[2] `https://github.com/embench/embench-iot`



(a) Slowdown of Qualcomm's Hexagon Simulator and VADL ISS relative to QEMU



(b) On a binary compiled without bundles, slowdown of hexagon-sim, non-VLIW VADL ISS, and VLIW VADL ISS relative to QEMU

Figure 5.1: Runtime of a benchmarks using Hexagon simulators (logarithmic scale, smaller is better)

CHAPTER 6

Future Work

There are three main areas for possible improvements that are apparent after this work: supporting VLIW architectures in the remaining VADL components, supporting the remaining features of common VLIW architectures, and improving the language design to make specifications easier to write.

In this work, we only added VLIW support for the ISS, meaning that the VADL assembler generator and compiler backend generator still don't support VLIW instructions. For the compiler backend, this would require translating the `group` expression into another representation (or code) as expected by the compiler.

For the assembler, more language constructs are likely needed: there are multiple ways of describing bundles in assembly (Hexagon uses curly brackets to group instructions together, while C6x uses `||` to join two surrounding instructions into a bundle). Hexagon's assembly syntax doesn't use mnemonics and is instead similar to higher-level languages with assignments and functions calls.

As already mentioned in the previous chapter, there is ongoing work to reimplement the ISS to achieve performance comparable with the existing mainstream simulators. The learnings from this work would have to be applied (and be reimplemented) there as well. One additional benefit of a JIT-based simulator is that the bundle parsing only has to happen once (before the native code is generated). It will also solve the general performance problems of an interpreted simulator and a naive decoder implementation (not a decoding tree but a linear search).

Some architectural features that are currently impossible to model with the VADL language (or not fully implemented). For C6x, this would be the exposed delay slots (which are supported in the parser, but not in any backend). Both C6x and Hexagon support a compressed version of instruction set (TI calls them compressed and Qualcomm calls them duplex instructions). One potential solution for this would be introducing a preprocessing function that can unpack and reorder (i.e. decompress) such bundles into

their equivalent representation. Hexagon has a number of other features that also operate on the whole bundle or rather require interaction between individual instructions (new-value instructions and hardware loops). Potentially all of these aspects could be solved with such a preprocessing function. The VADL parser does also support declaring loop registers and attaching a function to execute control logic.

Finally, being able to specify constraints on **`operation`**s could make specifications easier to write and a more domain-specific constraint language (replacing the nested quantified first order logic) could be more readable and more efficient (in the worst case with many nested quantified expression).

There were also a few cases where the VADL language itself made specifying the Hexagon ISA more complicated than perhaps necessary (unrelated to it being a VLIW architecture). Ideally, a macro would be able to create the two slightly different instruction format definitions which are required by most instructions (as seen in listing 5.4). Even aside from constant extenders, the Hexagon specification required an unusually high number of format fields (relative to the instruction count), because many instructions just vary slightly regarding which bits are immediates and opcodes.

CHAPTER 7

Conclusion

In this work, we extended VADL to support specifying VLIW architectures and enabled the instruction set simulator to execute VLIW instructions. This required several changes both in the language frontend and in the simulator generator.

For the VADL language itself, support for `group` extended regular expressions was added. Additional arbitrarily dynamic constraints can be specified using annotations, which are based on the expressions of the imperative part of the language but notably also include (existential and universal) quantifiers and additional operators to read and compare instructions in a bundle with the operation sets. These constraints can also not only compare the types of instructions but also access the format fields of each instruction.

The biggest part of this work was the development of an algorithm to turn the extended regular expressions of `group` expressions into a deterministic automaton which can be executed efficiently in the simulator to determine bundle boundaries. This was achieved by augmenting deterministic finite automatons with counters and conditions, which allows modelling repetition and permutation elements not with an exponential number of states, but using a single numeric counter.

After a bundle was detected, it still has to be executed with the correct semantics. This was achieved by executing the bundle in two phases: first all operands and all memory locations are read, and only subsequently are results written back into registers and memory. This way, the hardware's parallel reads and parallel write-backs can be emulated on a sequential machine.

Finally, the language features and the VLIW simulator were evaluated qualitatively and quantitatively. The core VLIW aspects of the Qualcomm Hexagon and Texas Instruments C6x architectures could be specified and binaries executed. Both of these architectures have, however, unusual features which prevent their full specification in VADL. Performance measurements showed that the implemented simulator has a certain overhead, which makes the already slow VADL ISS somewhat slower still. This will however be solved by a QEMU backend which is currently being worked on.

Overview of Generative AI Tools Used

No AI tools were used during the creation of this work.

Bibliography

[ACT10] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. *Regular Expressions, au point*. Oct. 2010. arXiv: 1010.2604 [cs.FL]. URL: https://arxiv.org/pdf/1010.2604.pdf.

[AMR00] Shail Aditya, Scott A. Mahlke, and B. Ramakrishna Rau. „Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats“. In: *ACM Transactions on Design Automation of Electronic Systems* 5.4 (Oct. 2000), pp. 752–773. ISSN: 1084-4309. DOI: 10.1145/362652.362658. URL: https://doi.org/10.1145/362652.362658.

[Ant96] Valentin Antimirov. „Partial derivatives of regular expressions and finite automaton constructions“. In: *Theoretical Computer Science* 155.2 (1996), pp. 291–319. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(95)00182-4. URL: https://www.sciencedirect.com/science/article/pii/0304397595001824.

[ARR00] S. Aditya and B. Ramakrishna Rau. *Automatic Architecture Synthesis and Compiler Retargeting for VLIW and EPIC Processors*. Tech. rep. Hewlett-Packard Company, 2000.

[ARRK99] S. Aditya, B. Ramakrishna Rau, and V. Kathail. „Automatic architectural synthesis of VLIW and EPIC processors“. In: *Proceedings 12th International Symposium on System Synthesis*. 1999, pp. 107–113. DOI: 10.1109/ISSS.1999.814268.

[Arsa] „Itanium's demise approaches: Intel to stop shipments in mid-2021“. In: *Ars Technica* (1st Feb. 2019). URL: https://arstechnica.com/gadgets/2019/02/itaniums-demise-approaches-intel-to-stop-shipments-in-mid-2021/ (visited on 08. 07. 2024).

[Arsb] „Microsoft: it's the end of the line for Itanium support“. In: *Ars Technica* (5th Apr. 2010). URL: https://arstechnica.com/information-technology/2010/04/microsoft-its-the-end-of-the-line-for-itanium-support/ (visited on 08. 07. 2024).

[BBR00] I. Barbieri, M. Bariani, and M. Raggio. „A VLIW architecture simulator innovative approach for HW-SW co-design“. In: *2000 IEEE International Conference on Multimedia and Expo. ICME2000. Proceedings. Latest Advances in the Fast Changing World of Multimedia (Cat. No.00TH8532)*. Vol. 3. 2000, pp. 1375–1378. DOI: 10.1109/ICME.2000.871022.

[BC08] Michela Becchi and Patrick Crowley. „Extending finite automata to efficiently match Perl-compatible regular expressions“. In: *Proceedings of the 2008 ACM CoNEXT Conference*. CoNEXT '08. Madrid, Spain: Association for Computing Machinery, 2008. ISBN: 9781605582108. DOI: 10.1145/1544012.1544037. URL: https://doi.org/10.1145/1544012.1544037.

[BKW98] Anne Brüggemann-Klein and Derick Wood. „One-Unambiguous Regular Languages“. In: *Information and Computation* 140.2 (1998), pp. 229–253. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1997.2688. URL: https://www.sciencedirect.com/science/article/pii/S0890540197926882.

[Bra+04] Gunnar Braun et al. „A novel approach for flexible and consistent ADL-driven ASIP design“. In: *Proceedings of the 41st annual Design Automation Conference*. DAC04. ACM, June 2004, pp. 717–722. DOI: 10.1145/996566.996763. URL: http://dx.doi.org/10.1145/996566.996763.

[Brz64] Janusz A. Brzozowski. „Derivatives of Regular Expressions“. In: *J. ACM* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: 10.1145/321239.321249. URL: https://doi.org/10.1145/321239.321249.

[Cod+14] Lucian Codrescu et al. „Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications“. In: *IEEE Micro* 34.2 (2014), pp. 34–43. DOI: 10.1109/MM.2014.12.

[Dan] *Guide to RISC Processors: for Programmers and Engineers*. Springer Science+Business Media, Inc, 2005, pp. 97–120. ISBN: 978-0-387-27446-1. DOI: 10.1007/0-387-27446-4_7.

[FFY05] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 9780080477541.

[Fre+25] Florian Freitag et al. *The Vienna Architecture Description Language*. Feb. 2025. arXiv: 2402.09087 [cs.PL]. URL: https://arxiv.org/abs/2402.09087.

[FVF95] Andreas Fauth, Johan Van Praet, and Markus Freericks. „Describing instruction set processors using nML“. In: *Proceedings the European Design and Test Conference. ED&TC 1995*. IEEE. 1995, pp. 503–507. DOI: 10.1109/EDTC.1995.470354.

[Glu61] Victor Mikhaylovich Glushkov. „The abstract theory of automata“. In: *Russian Mathematical Surveys* 16.5 (1961), p. 1.

[HHD00] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. „ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration“. In: *Design Automation for Embedded Systems* 6.1 (2000), pp. 39–69. DOI: 10.1023/A:1008937425064. URL: https://doi.org/10.1023/A:1008937425064.

[Hov09] Dag Hovland. „Regular Expressions with Numerical Constraints and Automata with Counters“. In: *Theoretical Aspects of Computing - ICTAC 2009*. Ed. by Martin Leucker and Carroll Morgan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 231–245. ISBN: 978-3-642-03466-4.

[Hov12] Dag Hovland. „The Membership Problem for Regular Expressions with Unordered Concatenation and Numerical Constraints“. In: *Language and Automata Theory and Applications*. Ed. by Adrian-Horia Dediu and Carlos Martín-Vide. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 313–324. ISBN: 978-3-642-28332-1.

[HPF13] Mian-Muhammad Hamayun, Frédéric Pétrot, and Nicolas Fournel. „Native simulation of complex VLIW instruction sets using static binary translation and Hardware-Assisted Virtualization“. In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2013, pp. 576–581. DOI: 10.1109/ASPDAC.2013.6509660.

[Käs03] Daniel Kästner. „TDL: A Hardware Description Language for Retargetable Postpass Optimizations and Analyses“. In: *Generative Programming and Component Engineering*. Ed. by Frank Pfenning and Yannis Smaragdakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 18–36. ISBN: 978-3-540-39815-8.

[Lei19] Steven Leibson. „News Flash: Itanic Still Sinking“. In: *EE Journal* (13th Feb. 2019). URL: https://www.eejournal.com/article/news-flash-itanic-still-sinking/ (visited on 08. 07. 2024).

[Liu+05] Chia-Hsien Liu et al. „Hierarchical instruction encoding for VLIW digital signal processors“. In: *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2005, 3503–3506 Vol. 4. DOI: 10.1109/ISCAS.2005.1465384.

[Mana] *Hexagon V60/V61 Programmer's Reference Manual*. 80-N2040-33 Rev. D. Qualcomm Technologies, Inc. 2019.

[Manb] *Intel Itanium Architecture - Software Developer's Manual - Volume 1: Application Architecture*. Rev. 2.3. Intel. 2010.

[Manc] *TMS320C64x/C64x+ DSP CPU and Instruction Set - Reference Guide*. SPRU732J. Texas Instruments Incorporated. 2010.

[MFP12] Luc Michel, Nicolas Fournel, and Frédéric Pétrot. „Fast simulation of systems embedding VLIW processors“. In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '12. Tampere, Finland: Association for Computing Machinery, 2012, pp. 143–150. ISBN: 9781450314268. DOI: 10.1145/2380445.2380472. URL: https://doi.org/10.1145/2380445.2380472.

[MM97] Jaime H. Moreno and Mayan Moudgil. „Scalable instruction-level parallelism through tree-instructions“. In: *Proceedings of the 11th International Conference on Supercomputing*. ICS '97. Vienna, Austria: Association for Computing Machinery, 1997, pp. 1–11. ISBN: 0897919025. DOI: 10.1145/263580.263584. URL: https://doi.org/10.1145/263580.263584.

[Mor+97] J. H. Moreno et al. „Simulation/evaluation environment for a VLIW processor architecture“. In: *IBM Journal of Research and Development* 41.3 (1997), pp. 287–302. ISSN: 0018-8646. DOI: 10.1147/rd.413.0287.

[P+11] Zdeněk Přikryl et al. „Design and Simulation of High Performance Parallel Architectures Using the ISAC Language“. In: *GSTF International Journal on Computing* 1.2 (2011), pp. 97–106. ISSN: 2010-2283. DOI: 10.5176/2010-2283_1.2.46. URL: https://www.fit.vut.cz/research/publication/9519.

[Pee+99] Stefan Pees et al. „LISA – machine description language for cycle-accurate models of programmable DSP architectures“. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. 1999, pp. 933–938. URL: https://dl.acm.org/doi/pdf/10.1145/309847.310101.

[Res+03] Mehrdad Reshadi et al. „An efficient retargetable framework for instruction-set simulation“. In: *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '03. Newport Beach, CA, USA: Association for Computing Machinery, 2003, pp. 13–18. ISBN: 1581137427. DOI: 10.1145/944645.944649. URL: https://doi.org/10.1145/944645.944649.

[RGD06] Mehrdad Reshadi, Bita Gorjiara, and Nikil D. Dutt. „Generic Processor Modeling for Automatically Generating Very Fast Cycle-Accurate Simulators“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.12 (2006), pp. 2904–2918. ISSN: 1937-4151. DOI: 10.1109/TCAD.2006.882597.

[RS59] M. O. Rabin and D. Scott. „Finite Automata and Their Decision Problems“. In: *IBM Journal of Research and Development* 3.2 (1959), pp. 114–125. ISSN: 0018-8646. DOI: 10.1147/rd.32.0114.

[Tho68] Ken Thompson. „Programming Techniques: Regular expression search algorithm“. In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: https://doi.org/10.1145/363347.363387.

[Tur+20] Lenka Turoňová et al. „Regex matching with counting-set automata“. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428286. URL: https://doi.org/10.1145/3428286.

[WYW08] Peng Wang, Jianxin Yang, and Biao Wang. „Simple-VLIW: A fundamental VLIW architectural simulation platform“. In: *2008 Asia Simulation Conference - 7th International Conference on System Simulation and Scientific Computing*. 2008, pp. 1258–1266. DOI: 10.1109/ASC-ICSC.2008.4675563.

[XL23] Xin Xiao and Zhong Liu. „ISADL: An Instruction Set Architecture Description Language for VLIW“. In: *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 2023, pp. 92–99. DOI: 10.1109/ICPADS60453.2023.00022.

[Yan+16] Lei Yang et al. „An approach to build cycle accurate full system VLIW simulation platform“. In: *Simulation Modelling Practice and Theory* 67 (2016), pp. 14–28. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2016.06.006. URL: https://www.sciencedirect.com/science/article/pii/S1569190X16301010.

[ZHS10] Zhe Zhang, Xiaoming Hu, and Linxiang Shi. „High-performance instruction-set simulator for TMS320C62x DSP“. In: *2010 The 2nd International Conference on Industrial Mechatronics and Automation*. Vol. 1. 2010, pp. 517–520. DOI: 10.1109/ICINDMA.2010.5538061.