Generation of scalar ASIPs with OpenVADL

Linus Halder
Compilers and Languages Group
Technische Universität Wien
Vienna, Austria
vadl@tuwien.ac.at

Matthias Raschhofer
Compilers and Languages Group
Technische Universität Wien
Vienna, Austria
vadl@tuwien.ac.at

Andreas Krall
Compilers and Languages Group
Technische Universität Wien
Vienna, Austria
vadl@tuwien.ac.at

***Abstract*—OpenVADL is a collaborative open source implementation of the Vienna Architecture Description Language (VADL). VADL is a processor description language (PDL) that enables the concise formal specification of processor architectures. VADL strictly separates the specification of the instruction set architecture (ISA) from the micro architecture (MiA). Until now OpenVADL only supported the generation of a QEMU based instruction set simulator and the generation of LLVM based assemblers, linkers and compilers. This work presents the so far missing architecture synthesis derived from a MiA specification and the generation of the complete processor in the hardware description language Chisel. Architecture synthesis uses the instruction progress graph (IPG). The IPG combines the behavior of all instructions distributed to the different pipeline stages. The evaluation demonstrates that the fully automatically generated RISC-V processors are comparable in area and frequency to manually specified cores like Sodor and Wildcat.**

***Index Terms*—ASIP, processor description language, automated processor generation, VADL, Chisel, RISC-V.**

I. Introduction

The design, implementation, testing and evaluation of Application Specific Instruction Set Processors (ASIPs) is a complex and laborious process. The use of a Processor Description Language (PDL) dramatically reduces the development effort. The Vienna Architecture Description Language (VADL) is a PDL that enables the concise formal specification of processor architectures [1]. VADL strictly separates the specification of the Instruction Set Architecture (ISA) from the Microarchitecture (MiA). Two years ago, a collaborative effort to develop an open-source implementation of VADL, Open Source VADL (OpenVADL), was started and is available on GitHub [1] [2]. Until now OpenVADL did not support the generation of a processor description in a hardware specification language.

This work adds micro architecture and hardware synthesis to OpenVADL. The novel idea of an Instruction Progress Graph (IPG) is presented. The IPG is used to map and combine the behavior of single instructions to the pipeline stages of a processor. Section II introduces VADL. Section III describes the overall architecture of OpenVADL. Section IV presents the VADL Decode Tree (VDT) and its usage in decoder generation. Section V explores the IPG and its optimizations for hardware generation. Section VI gives a detailed evaluation of the generated processor.

[1]https://github.com/OpenVADL/openvadl

II. The Vienna Architecture Description Language

VADL follows the design principle of the separation of concerns and strictly separates the specification of the ISA and the MiA of a processor architecture. Therefore, VADL specifications contain different sections describing the ISA and the MiA. Further sections, such as the Application Binary Interface (ABI) for the compiler and the assembly description for an assembly language parser, are not discussed in this work.

A. Instruction Set Architecture Definition

Listing 1 shows a complete ISA specification of all RISC-V load instructions. To ease the use of a single specification for different architecture sizes a constant for the architecture size is defined in line 3. Lines 5 to 10 declare type aliases for VADLs bit vector types. The basic type is `Bits`. There exist two subtypes representing signed (`SInt`) and unsigned (`UInt`) two's complement integers. Line 13 demonstrates the definition of a register file where the register with index 0 is always zero. Annotations (similar to the zero annotation in line 12) can be used to detail a definition. Similar to the register definition the memory definition maps an address range to a memory element (line 14). An implicitly updated program counter is required in every ISA specification (line 15). A format definition is used to specify bitfields with named and typed member fields (line 17 to 24). Field access functions specify computations on format fields (line 23).

An instruction requires the specification of the instruction's behavior, encoding and assembly (lines 27 to 33). The instruction `MAXI` uses the instruction format `Itype`. The behavior is specified in a functional programming style. Registers and memory can only be written once and all reads must occur before a write. The type cast `as URe`g zero extends the format field `imm` to a 32-bit vector.

Usually, many instruction definitions are quite similar. VADL supports type safe syntactic macro templates to avoid copying and modifying specifications. A macro definition starts with the keyword `model` followed by the typed arguments and the result type of the macro. There exist syntactic types like `Id` (identifier), `Bin` (binary constant), `CallEx` (subtype of the expression type `Ex`) or `IsaDefs` (ISA definitions). An instantiation of a macro or the substitution of a macro argument are indicated by the dollar sign.

```
1  instruction set architecture RV32I = {
2
3  constant Size = 32          // architecture size 32 bits
4
5  using Byte    = Bits<  8 >  // 8 bit Byte
6  using Inst    = Bits< 32 >  // instruction word type
7  using UReg    = Bits<Size>  // register word type
8  using SReg    = SInt<Size>  // signed register word type
9  using Index   = Bits<  5 >  // 5 bit register index type
10 using Addr    = UReg        // address equal register size
11
12 [zero : X(0)]               // X(0) always is zero
13 register  X : Index -> UReg // 32 integer registers
14 memory  MEM : Addr  -> Byte // byte addressed memory
15 program counter PC : Addr   // PC points to instruction
16
17 format Itype : Inst =       // immediate instruction format
18   { imm      : Bits<12>     // [31..20] immediate operand
19   , rs       : Index        // [19..15] source register
20   , funct3   : Bits<3>      // [14..12] 3 bit function code
21   , rd       : Index        // [11..7]  destination register
22   , opcode   : Bits<7>      // [6..0]   7 bit operation code
23   , immS     = imm as SReg  // sign extended immediate
24   }
25
26 // fictitious unsigned max immediate instruction
27 instruction MAXI : Itype =
28   let xrs  = X(rs) in
29   let immU = imm as UReg in
30     X(rd) := if xrs > immU then xrs else immU
31 encoding MAXI = {opcode = 0b001'0011, funct3 = 0b001}
32 assembly MAXI = (mnemonic, " ", register(rd), ",",
33                                 register(rs), ",", udec(imm))
34
35 // macro for RV32I load instructions
36 model L_Instr (name:Id,fun3:Bin,memEx:CallEx,exTy:Id):IsaDefs={
37   instruction $name : Itype =
38     let addr = X(rs) + immS in
39       X(rd) := $memEx as $exTy
40   encoding $name = {opcode = 0b000'0011, funct3 = $fun3}
41   assembly $name = (mnemonic, " ", register(rd), ",",
42                     decimal(imm), "(", register(rs), ")")
43   }
44
45 $L_Instr (LB ;0b000;MEM   (addr);SReg) // load byte signed
46 $L_Instr (LBU;0b100;MEM   (addr);UReg) // load byte unsigned
47 $L_Instr (LH ;0b001;MEM<2>(addr);SReg) // load half signed
48 $L_Instr (LHU;0b101;MEM<2>(addr);UReg) // load half unsigned
49 $L_Instr (LW ;0b010;MEM<4>(addr);SReg) // load word
50 }
```

Listing 1. ISA specification of all RISC-V RV32 load instructions

By packing the instruction, encoding and assembly definition into a macro, a load instruction can be specified in a single line (lines 36 to 43). This macro is invoked five times for all RISC-V load instructions (lines 45 to 49).

B. Micro Architecture Definition

The MiA section contains the necessary specifications to map different parts of an instruction's behavior to the corresponding architectural parts of a processor. The main definition element is a pipeline stage and multiple dependent pipeline stages form a complete pipeline. Pipeline stages which execute similar behavior for a subset of the defined instructions can be seen as functional units. The order of pipeline stages in a pipeline is implicitly determined by references to other pipeline elements.

The MiA specification relies on the fact that the behavior of instructions have some implicit common properties. Examples for this kind of implicit properties are the decoding of instructions, accessing immediate instruction operands, transforming immediate operands, reading or writing registers or memory, doing computations or computing addresses.

There exist two very high level type abstractions for MiA elements. The type `FetchResult` abstracts all information which results from fetching instructions from memory (see line 7 from Listing 2). The type `Instruction` abstracts all the varying information which results from instructions traveling through the pipeline (see line 10 or 17 from Listing 2).

Functions and methods defined on these two types allow the extraction of relevant data based on the implicit properties. For example, the function `decode` takes a value of type `FetchResult` and converts it into a value of type `Instruction` (see line 11). The method `readOrForward` of an `Instruction` typed value either reads a value from a register or consumes a forwarded value (see line 12). The method `compute` identifies all the computations on the right hand side of an assignment to a register or memory (see line 19).

Some additional MiA elements, such as caches, buffers, and queues for superscalar execution, are currently not supported.

```
1  import isa::{RV32I}
2
3  micro architecture FiveStage implements RV32I = {
4
5  logic [forwarding] bypass
6
7  stage FETCH -> (fr : FetchResult) = {
8    fr := fetchNext      // fetch next packet from memory (cache)
9    }
10 stage DECODE -> (ir : Instruction) = {
11   let instr = decode(FETCH.fr) in {  // decode fetch packet
12     instr.readOrForward(@X, @bypass) // read from X registers
13     instr.read(@PC)    // read from the PC
14     ir := instr        // stage output is ir of type Instruction
15     }
16   }
17 stage EXECUTE -> (ir : Instruction) = {
18   let instr = DECODE.ir in {
19     instr.compute      // evaluate all expressions
20     instr.results(@X, @bypass) // forward results of compute
21     instr.verify       // flush pipeline for mispredicted branch
22     instr.address(@MEM)// compute memory address
23     instr.write(@PC)   // write PC
24     ir := instr
25     }
26   }
27 stage MEMORY -> (ir : Instruction) = {
28   let instr = EXECUTE.ir in {
29     instr.write(@MEM)  // write to memory
30     instr.read(@MEM)   // receive data from memory read
31     instr.results(@X, @bypass) // forward results of read
32     ir := instr
33     }
34   }
35 stage WRITE_BACK = {
36   let instr = MEMORY.ir in {
37     instr.write(@X)    // write back to register file X
38     instr.results(@X, @bypass) // forward results of write
39     }
40   }
41 }
```

Listing 2. MiA specification of a simple five stage pipeline

The two presented ISA and MiA examples only touch the surface of the rich and complex language. A detailed description of the language is available in the OpenVADL documentation[2] and in [1].

[2]https://openvadl.github.io/openvadl/

III. OpenVADL Implementation

A. Overview

Figure 1 gives an overview of the current OpenVADL architecture. The frontend translates a VADL processor specification into the VADL Intermediate Architecture Model (VIAM). The VIAM is then used by the generators to produce the LLVM based compiler, assembler and linker and the QEMU based instruction set simulator. This work adds the architecture synthesis which maps all parts of an instruction's behavior to the correct MiA elements represented in the IPG. The IPG is the basis for the hardware generator and may be used in the future to generate a cycle approximate simulator or generate instruction scheduling information for the compiler.

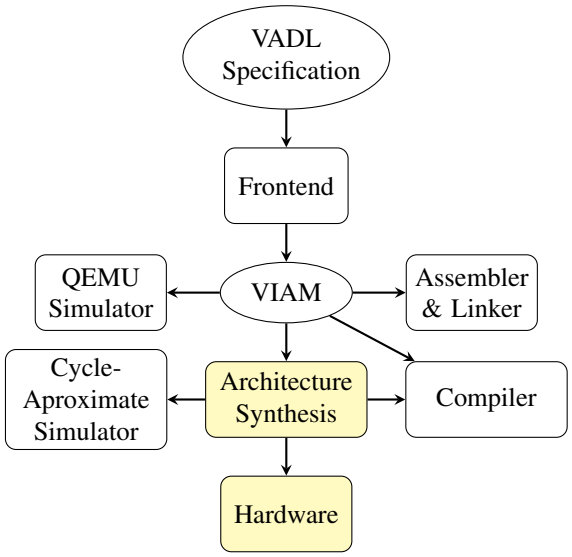

Fig. 1. Overview of the OpenVADL architecture. Contributions of this work are highlighted in yellow.

B. VIAM

The VIAM contains the complete information about a processor including all data required by the different generators. Most definitions like registers or MiA elements are simply represented as lists or maps of data structures. Definitions that encapsulate behavior, such as instructions and functions, include a graph representing their behavior, also referred to as the VIAM behavior graph. The VIAM behavior graph is a multigraph that includes both a Control Flow Graph (CFG) and a Data Dependency Graph (DDG) representing a Program Dependence Graph (PDG). As VADL neither supports loops nor recursion all graphs are Directed Acyclic Graphs (DAGs). The generators use graph traversal algorithms that operate in both directions, namely from sources to sinks and from sinks to sources. Accordingly, edges are bidirectional, and there are links from the definitions both to all sources and to all sinks. The VIAM graph in Figure 2 shows the edges in the data flow direction.

Control Flow Graph. The `MAXI` instruction in Figure 2 features a minimal CFG, consisting only of a start and end node. An if-else expression is represented by a select node. Select nodes can have multiple inputs and multiple outputs and select the one of the inputs depending on the selection condition. If-else statements result in a more complex if-else control flow representing conditional side effects using additional start and end nodes.

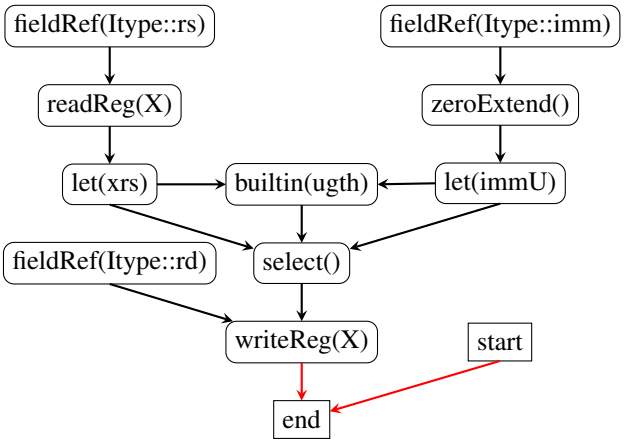

Fig. 2. Simplified VIAM behavior graph of the `MAXI` instruction, specified in Listing 1.

Data Dependency Graph. To improve readability, VADL behavior follows a sequential reading, while user constraints allow for parallel semantics. This ensures that all resource reads have completed before computations of an instruction begins. All writes take effect only after the computations have completed. This is feasible because VADL instructions cannot write to the same resource twice along the same execution path, preventing undefined behavior between reads and writes.

With parallel semantics, not only expressions but also side-effect nodes, such as register writes, can be incorporated into the dependency graph. As shown in Figure 2, resource writes are dependencies of a branch end node. If a write is conditional, the branch end node corresponds to the last node of the taken if-branch. Since most generators operate on dependency graphs, this structure enhances flexibility. Additionally, all dependency nodes are unique, ensuring that each side effect or expression appears exactly once in the graph—an inherent optimization. In later optimization phases the let nodes are eliminated, as not only the let nodes, but also every behavior graph node can have an unlimited number of successor nodes.

IV. Decoder Generation

Instructions specified in the ISA section consist of the definition of an instruction `format` to which the instruction conforms, as well as an `encoding` definition, specifying the fixed format fields for that particular instruction. From this specification, OpenVADL constructs an abstraction of the decoding procedure, as part of the VIAM.

The decoding procedure is modeled as a decision tree over the input bits of an encoded instruction. Accordingly, the generated intermediate representation is called the VADL Decode Tree. From this common intermediate representation,

generator-specific implementations of the decoder are derived. In particular, the implementation for the Instruction Set Simulator (ISS) differs from the hardware decoder presented in the context of this work.

A. Input preparation

The decode tree generation mainly operates on bit patterns. A bit pattern is a bit vector which may contain a wildcard in any bit position ($p \in \{0, 1, -\}^n$), and it can be efficiently implemented as the combination of two bit vectors. To this effect, the VIAM definitions of instruction formats and encodings are flattened to a set of bit patterns, consisting of the assigned values in the encoding definition, and wildcards in the position of instruction operands.

In addition to the opcode values assigned in the encoding definition, VADL is extended to support the specification of encoding constraints. Such constraints are additional logical expressions over fields of the instruction format, to enable the specification of irregular encoding schemes. Concretely, this allows the definition of encodings with different lengths, as well as overlapping static and dynamic fields.

The `select when` annotation takes an expression in a restricted fragment of quantifier-free predicate logic with equality, to specify the encoding constraints for selecting an instruction, and to eliminate any ambiguities. This expression is transformed to a normal form and subsequently to a set of conditional bit patterns during a preparation pass. In the form of bit patterns, the constraints are passed as additional input to the decision tree generation.

To ease the specification effort, some encoding constraints are synthesized automatically during input preparation. In particular, constraints for subsumed instructions can be constructed automatically by giving precedence to more specific instruction encodings, which will be selected over the more general case. A formal verification pass ensures that no further collisions or ambiguities between instruction encodings exist beyond this step.

B. Construction of the decision tree

The inner nodes of the VDT represent a decision function, and the leaf nodes describe the instructions to be decoded by matching all functions on the path from the root node. To be able to efficiently implement the decode tree in hardware, the decision functions are restricted to basic bitwise operations and comparisons.

The generation algorithm is based on [3] and [4], and operates on a set of instruction encoding patterns, which are recursively partitioned into subsets by selecting certain decision functions. The sub trees constructed through this divide and conquer approach are recursively combined to construct a single decision tree.

Finding a divisive decision function is a search problem in itself [5], and the algorithm implemented in this work handles it in two ways:

- Constructing a bit mask from the statically known values in the encoding pattern (usually opcodes).
- Selecting splitting patterns based on encoding constraints.

These decision functions result in n-ary and binary decisions respectively. As shown by [4], custom encoding constraints can be used effectively to construct binary decision functions during decision tree generation. Note that this strategy may cause multiple leaves in the decision tree to be labeled with the same instruction, as there may be multiple decision paths resulting in the same conclusion.

Note also that, due to the way the decision functions are selected, the generation algorithm underapproximates the set of all valid encoding schemes. However, successfully applying the approach to a variety of instruction set architectures shows that this is not a problem in practice [4]. Moreover, the algorithm can be guided by adding additional constraints.

C. Integration to the Hardware Description

Throughout the architecture synthesis, the decoder is only present as an abstract node in the IPG. It is mapped to a MiA stage according to the *decode* built-in call, and subsequently used as input to select nodes in the IPG.

Since the VDT only describes the decision procedure, not the behavior in case an instruction is matched (or an invalid instruction is detected), this has to be defined before emitting a concrete implementation of the decoder in a Hardware Description Language (HDL). To achieve this, the usage of the abstract decode node in the constructed MiA implementation and the instruction annotations of the select-nodes are analyzed. Each usage is extracted as a decision signal, and the instruction annotations of the select-nodes determine the values to be emitted depending on the decoded instruction.

These decode signals are collected, and combined with the HDL representation of the decision procedure described by the VDT as well as the constructed MiA implementation during emission of the hardware description.

V. Processor Generation

Starting from the VIAM provided by the frontend the processor generator takes a MiA definition and produces a hardware description from it. This process is divided into two steps: the architecture synthesis and the generation of the hardware description output. The architecture synthesis transforms the MiA definition into a concrete implementation. The abstract MiA constructs used to fetch, decode and execute instructions are replaced by implementations derived from the ISA behavior. In addition to the stage definitions present, this also creates control and forwarding logic in order to stall stages where hazards cannot be resolved. This produces a MiA implementation represented in the VIAM and can be used by other generators, e.g., for information on forwarding conditions and better estimations of the stalling behavior and utilization of the pipeline. The central data structure for the architecture synthesis is the Instruction Progress Graph (IPG) described in the following.

A. Instruction Progress Graph

The IPG contains the behavior of all instructions combined into a single graph. It is used to represent the intermediate states of instructions during their execution implemented in the stages of the MiA. Each node is annotated with the set of instructions it occurs in. The nodes are also associated with the stage it will be included in when constructing the resulting MiA implementation. In order to build this graph, first, all instructions are preprocessed by eliminating the control flow. The conditions necessary for a read or write to be reached in the control flow are added as condition inputs to the read and write nodes, turning the control flow into a data flow representation. Control flow nodes are removed and all dependency nodes are copied from each instruction to the IPG while annotating all copied nodes with the processed instruction.

The VIAM graph implementation supports deduplicating nodes on insertion and this is used to recognize behavior shared between instructions, resulting in nodes annotated with multiple instructions where possible. An example for this is given in Figure 3. The behavior shared between the `MAXI` and `LW` instructions is annotated with both instructions. The `MAXI` instruction writes back the greater of two values, a register value and an immediate value, to the destination register. `LW` calculates an address by adding an immediate value to a register value and loads the word at this address from memory to the destination register.

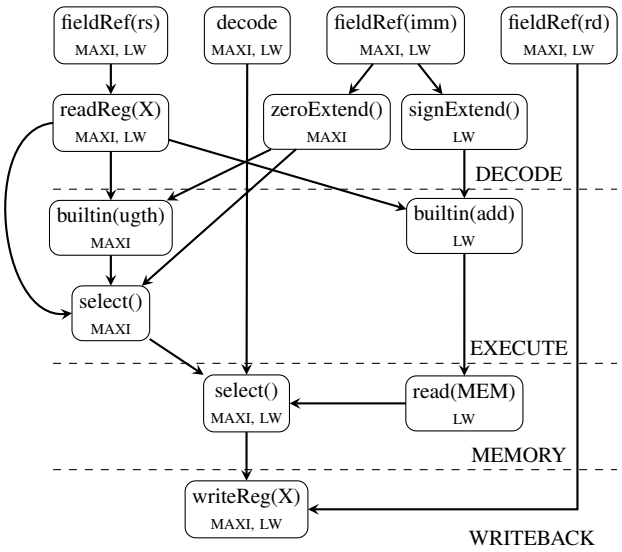

Fig. 3. Simplified IPG of the `MAXI` and `LW` instruction. Nodes are annotated with the instructions the node belongs to. Dashed lines indicate the partition of the IPG in subgraphs for each MiA stage. The MiA specification for the five-stage pipeline used here is from Listing 2 described in Section II.

Behavior that is common to all instructions and that can be mapped to a stage, is also added to the IPG. This mainly concerns generating an implementation for the *fetchNext* built-in, taking care of reading an instruction word from memory at the address stored in the program counter and incrementing it, or alternatively use a branch predictor to predict the next program counter value.

For associating the IPG nodes with the stages, a linear order of the stages is determined by examining the data dependencies between the stages. Then we iterate over the stages in this order and examine the built-in calls present, each of them is mapped to a selection of nodes from the IPG. The selection of nodes and all its dependencies not already mapped to a previous stage are associated with the current stage using the IPG data structure. These nodes are selected based on the built-in call and the current assignment of nodes within the IPG:

- *fetchNext*: generated implementation with the instruction word read and next program counter write
- *decode*: maximal set of nodes such that all nodes (i) do not depend on read nodes and (ii) are not writes
- *compute*: all built-in calls that operate on more than one bit wide inputs
- *read(@R)*: all reads from the resource R
- *write(@R)*: all writes to the resource R

Note that the rule for the *decode* built-in implies that all nodes only dependent on the instruction word are associated with the stage that contains the decode built-in.

In general some of the IPG nodes can not be selected with these rules, e.g., select nodes. All such nodes are associated to a stage solely because they are a dependency of a node that was selected by a rule. These nodes are treated in a particular manner. They are not considered to be fixed to the specific stage and may be reassigned to adjacent stages as long as the data dependencies permit it. Optimization passes utilize this to reduce the number of stage output registers, or in terms of the IPG, to reduce the number of inter-stage dependencies.

After the built-in calls are processed in all stages, it is ensured that every IPG node is associated with a stage, making sure that all instruction behavior will be implemented. Figure 3 illustrates how the result of this step (the association of IPG nodes to stages) forms subgraphs on the IPG.

In order to simplify the behavior graphs and the later generation of the hardware description the read and write nodes referencing the same resources are merged in each stage. This is done by checking that they are not annotated with overlapping instructions and therefore cannot be active simultaneously. The inputs of the merged nodes are replaced with select nodes that are controlled by the output of the decoder detecting which instruction we actually execute. Similar to the select nodes the condition inputs of the read and write nodes are masked with decoder outputs that are only true, if an instruction that the read/write node is annotated with was decoded.

The generator currently does not optimize the arithmetic operations in the IPG. In a standard five-stage pipeline this results in an execute stage containing all possible arithmetic and logic operations with select nodes choosing the actual result. Optimizing this is left to the synthesis tool. Since it is not guaranteed that the tool will produce an optimal implementation, this is an area for future study and improvement.

The IPG is now in a state that can directly be inlined into the stages in the VIAM. Subgraphs following from the stage annotations in the IPG are copied to the respective stages. For every output to a node outside the stage's subgraph stage registers are created that hold the intermediate values during an instruction's execution. For every missing input the appropriate stage register of the previous stage is read. The result is a preliminary MiA implementation without forwarding or control logic.

B. Forwarding and Control Logic

The key component for generating control logic is an analysis of the data hazards in the pipeline. It results from considering all pairs of reads and writes on the same resource where the read occurs in an earlier stage than the write. For these pairs all stages between the read and write need to be analyzed for the intermediate value of the write condition and address. If any of the values is not available this situation is treated as a hazard. To inform forwarding the analysis also assesses, if the value is available in an intermediate stage.

Writes to the program counter are an exception, a hazard is not detected until the write condition is true and a value is available to allow speculative execution.

The IPG is used to evaluate if an input to a write is available in a given stage. This is done by traversing select nodes up in the dependency graph while collecting the select nodes conditions under which the input is available in a previous stage.

Forwarding logic is generated based on the MiA definition. If forwarding is not specified in the MiA, the generator assumes bypass on all feasible paths. To represent forwarding in the VIAM the forwarding logic element is extended with behavior that connects necessary signals from the stages to detect forwarding conditions and values. They are connected to the read nodes receiving forwarded values. The read node outputs are replaced by a select, choosing between the read value and the forwarded value.

The control logic consists of registers tracking if there is an instruction in a stage, enable signals for each stage and internal stall signals, it is based on [6]. Data hazards are detected based on the results of the hazard analysis and the signals from the forwarding logic. External stalls from memory are connected to the stall logic. Each stage's reads and writes are masked with the enable signal. When a write to the program counter occurs apart from the speculative write during instruction fetch, all previous stages are rolled back. The resulting behavior is amended to the MiA definition in the form of a control logic element and this step yields the MiA implementation, the result of the architecture synthesis.

C. Generating the Hardware Description

In the constructed MiA implementation only constructs remain that can be emitted in a hardware description language (HDL). These are:

- Register files and registers defined for the ISA and inside the stage and logic element definitions
- Memory, implemented by connecting an external memory interface
- Behavior graphs inside the stages and logic elements

Register files are implemented as asynchronous memory with a read or write port for every stage that accesses it. The external memory interface also has a read port for instruction fetch and every memory read, as well as a write port for every memory write in the pipeline.

Stages and logic elements contain behavior graphs. They are broken down into statements implementing the graphs behavior. Wires are introduced to break up complex expressions and allow reusing the expression in multiple statements. Reads and writes are realized with statements connecting the appropriate signals of a read or write port.

The hardware generator currently emits Chisel code [7] using mostly basic constructs that should also map easily to other HDLs. These are modules having defined input and outputs, and containing registers, wires and conditional assignments to them. Chisel is an HDL embedded in Scala and generates Verilog code for simulation and synthesis. For register files the `Mem` construct is used. This generates Verilog that maps to register banks in modern technologies[3].

While Chisel was selected initially for being able to use Scala-based tests, the generated code does not use any Chisel-specific features and emitting Verilog directly remains an option for future work.

For testing in simulation the generated Chisel project is accompanied by a simple C++ Verilator model for memory. It can be used in Chisel tests and is able to load ELF files. We use this setup to test the generated hardware for the OpenVADL RISC-V specification against the RISC-V test suite[4].

VI. Evaluation

To demonstrate the use of hardware generation from a VADL specification, we compare the area and maximum clock frequency of synthesized RV32I cores that have a three and a five-stage variant. The existing cores selected for this are Sodor[5] and Wildcat [8]. They are each compared against a core generated from a VADL specification that follows the MiA of the variant as closely as possible using the constructs available. Specifically, we devised stages with reads and writes mapped and forwarding paths configured according to the core's documentations and HDL code. The implementation of control and status registers (CSRs) was removed from both cores, because they have very different ranges of features they implement.

The existing and the generated cores have been synthesized with the SkyWater Foundry SKY130 high density standard cell library using Yosys [9] and OpenROAD [10]. Our evalution uses an iterative setup to determine the individual maximum clock frequency of each design. Synthesis was configured with timing driven placement and 1 ns input and output delay.

[3]https://www.chisel-lang.org/docs/explanations/memories
[4]https://github.com/riscv-software-src/riscv-tests
[5]https://github.com/ucb-bar/riscv-sodor

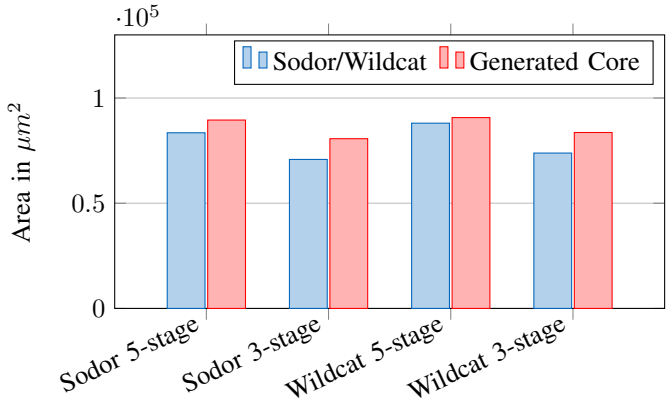

Fig. 4. Placement area of synthesized RV32I cores with the SkyWater130 ASIC process (lower is better).

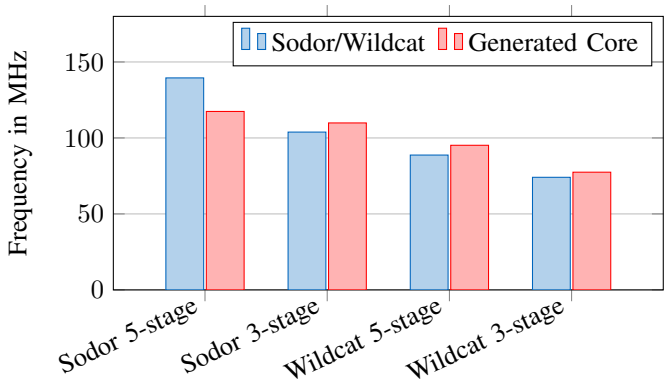

Fig. 5. Maximum clock frequency of synthesized RV32I cores with the SkyWater130 ASIC process (higher is better).

Figure 4 shows the area of the synthesized cores. All cores show similar area, with an overhead of around 10% for most of the generated cores.

Figure 5 shows the maximum clock frequency of the synthesized cores. Most cores have similar maximum clock frequency. The biggest difference exists between the Sodor five-stage implementation and the generated core, i.e., an about 20% higher clock speed for the Sodor core. The critical path in this generated core is the branch decision logic in the execute stage connected to control logic. This can potentially be improved by optimizing the generated comparison logic for the branch decision.

The generator operates at a level, where it is not able to optimize the timing of the design. Therefore, hand-written designs that are optimized for timing, like the five-stage Sodor core, can achieve better maximum frequencies.

On the other hand, the timing of the Sodor three-stage pipeline shows a case where the generated code trades area for performance. The critical path of the existing Sodor implementation is the address calculation for memory access through the common ALU shared by all instructions, while the generated core contains an additional adder for the address calculation. Note that this is a trade-off that currently cannot be expressed in the VADL specification.

To outline the effort in exploring different pipeline configurations using VADL, Table I gives the lines of code in the different design files used in the evaluation. A specification of the RV32I instruction set is available with OpenVADL. Changes to the MiA only require editing a rather small amount of code.

TABLE I
LINES OF CODE EXCLUDING COMMENTS AND WHITESPACE

		Sodor Pipeline		Wildcat Pipeline	
	Language	5-stage	3-stage	5-stage	3-stage
RV32I ISA	VADL	200	200	200	200
MiA	VADL	38	25	39	28
Generated Core	Chisel	1248	853	1282	945
Sodor/Wildcat	Chisel	823	724	452	425

Hardware description generation for the RISC-V five-stage configuration from its OpenVADL specification completes within 100-200 ms on an Apple MacBook Air (M2, 2022, 8-core CPU, 8 GB RAM). This allows for quick iteration during design-space exploration.

VII. Related Work

An extensive discussion of VADL and its original implementation is provided in [1], including a detailed comparison with other PDLs in terms of features and language properties. Existing work regarding the open-source implementation OpenVADL is presented in [2] and focuses on the QEMU based ISS and LLVM based compiler generators respectively.

The decoder generation presented in this work is based on [3], which describes the technique to generate decode decision trees from binary encodings and has since been adopted by many derivative works. To overcome the restriction to mostly regular encoding schemes, this work incorporates ideas from [4]. The article presents a novel approach to allow the specification of constraints in the form of exclusion patterns. The implementation in OpenVADL expands on this idea by automatically transforming user-defined constraints and synthesizing some constraints automatically.

A different approach to support irregular encodings is taken by [11]. The article describes the specification of encodings using Binary Decision Diagrams (BDDs), which can be used to construct a decision tree.

In addition, [11] describes a cost driven construction algorithm incorporating occurrence probabilities, also based on BDDs. A more generalised version of constructing decision trees purely from occurrence statistics is discussed in [5].

An alternative to generating decision trees for decoding is logic optimization from a table based specification. Chisel [7] provides built-in utilities to define decoders based on truth tables and optionally an optimal or heuristics based optimization algorithm [12] [13].

Hardware generation from a high-level MiA specification is a topic approached with different methods. ASSIST [14] allows the specification of instruction behavior and a pipeline

schedule. This schedule places a fixed set of datapath functional units that implement parts of the instruction behavior in stages, but with less granular control than the VADL MiA description. Additionaly, their tool supports autotuning the pipeline schedule, by iterative synthesis of the generated design.

PDL [15] proposes a pipeline description language that can describe stage behavior and uses operations on locks to prevent hazards. These locks reference register and memory addresses and are checked for correct usage in the pipeline. The lock implementations handle control, forwarding and speculation. Different from VADL, PDL does not separate between ISA and MiA and instruction behavior is specified inside the code of the stages.

An important part of hardware generation from a high-level description is the generation of control logic. OWL [16] synthesizes control logic for a datapath design using an ISA specification. The datapath is described on RTL, on a lower level of abstraction than the VADL MiA.

Producing hardware from instruction behavior without a fixed MiA through High-level synthesis (HLS), starting from an ISA specification in C, is explored in [17]. They transform the code of an ISS loop such that an HLS tool can infer a speculative pipeline.

VIII. Conclusion

With OpenVADL it is possible to develop competitive processors with linear pipelines in a dramatically reduced development time. A processor with the complexity of the RISC-V RV32I with multiple pipeline variants can be designed in less than a day. Adding an additional instruction like the `MAXI` instruction from section II is the matter of several minutes. The novel IPG and its optimizations are powerful techniques to generate efficient processors. In the future we will add additional optimizations and the support of non-linear and superscalar micro architectures. OpenVADL with processor generation as presented in this article is already available as open source on GitHub [6] and future improvements will be made available there.

References

[1] F. Freitag, L. Halder, S. Himmelbauer, C. Hochrainer, B. Huber, B. Kasper, N. Mischkulnig, M. Nestler, P. Paulweber, K. Per, M. Raschhofer, A. Ripar, T. Schwarzinger, J. Zottele, and A. Krall, "The Vienna Architecture Description Language," 2025. [Online]. Available: https://arxiv.org/pdf/2402.09087

[2] F. Freitag, L. Halder, B. Huber, B. Kasper, M. Nestler, K. Per, M. Raschhofer, A. Ripar, J. Zottele, and A. Krall, "OpenVADL: An open source implementation of the Vienna Architecture Description Language," in *ARCS 2025 – 38th GI/ITG International Conference on Architecture of Computing Systems*, S. Tomforde, C. Krupitzer, S. Vialle, E. Suarez, and T. Pionteck, Eds. Springer Nature Switzerland, 2025, pp. 156–171.

[3] H. Theiling, "Generating decision trees for decoding binaries," in *Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems*. ACM, 2001, pp. 112–120.

[4] K. Okuda and H. Takeyama, "Decision tree generation for decoding irregular instructions," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. EDAA, 2016, pp. 1592–1597. [Online]. Available: https://past.date-conference.com/proceedings-archive/2016/pdf/0066.pdf

[5] W. Qin and S. Malik, "Automated synthesis of efficient binary decoders for retargetable software toolkits," in *Proceedings of the 40th Annual Design Automation Conference*, ser. DAC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 764–769. [Online]. Available: https://doi.org/10.1145/775832.776027

[6] D. Kroening and W. J. Paul, "Automated pipeline design," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 810–815.

[7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[8] M. Schoeberl, "Wildcat: Educational RISC-V microprocessors," in *ARCS 2025 – 38th GI/ITG International Conference on Architecture of Computing Systems*, S. Tomforde, C. Krupitzer, S. Vialle, E. Suarez, and T. Pionteck, Eds. Springer Nature Switzerland, 2025, pp. 189–202.

[9] C. Wolf and J. Glaser, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013, p. 6. [Online]. Available: https://yosyshq.net/yosys/files/yosys-austrochip2013.pdf

[10] A. B. Kahng and T. Spyrou, "The OpenROAD project: Unleashing hardware innovation," in *Proc. GOMAC*, 2021, pp. 1–6.

[11] N. Fournel, L. Michel, and F. Pétrot, "Automated generation of efficient instruction decoders for Instruction Set Simulators," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, 2013, pp. 739–746.

[12] E. J. McCluskey, "Minimization of boolean functions," *The Bell System Technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[13] R. K. Brayton, G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli, "A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization," in *Proceedings of the International Symposium on Circuits and Systems*, 1982, pp. 42–48.

[14] G. Liu, J. Primmer, and Z. Zhang, "Rapid generation of high-quality RISC-V processors from functional instruction set specifications," in *Proceedings of the 56th Annual Design Automation Conference 2019*. Association for Computing Machinery, 2019, pp. 1–6.

[15] D. Zagieboylo, C. Sherk, G. E. Suh, and A. C. Myers, "PDL: a high-level hardware design language for pipelined processors," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 719–732.

[16] Z. D. Sisco, A. D. Alex, Z. Ma, Y. Aghamohammadi, B. Kong, B. Darnell, T. Sherwood, B. Hardekopf, and J. Balkind, "Control logic synthesis: Drawing the rest of the OWL," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2024, pp. 63–78.

[17] J.-M. Gorius, S. Rokicki, and S. Derrien, "Design exploration of RISC-V soft-cores through speculative high-level synthesis," in *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2022, pp. 1–6.

[6] https://github.com/OpenVADL/openvadl