OpenVADL

An open source implementation of the
Vienna Architecture Description Language

Florian Freitag, Linus Halder, Benedikt Huber
Benjamin Kasper, Michael Nestler, Kevin Per
Matthias Raschhofer, Alexander Ripar, Johannes Zottele

Andreas Krall

Compilers and Languages, TU Wien

April 24th 2025

Outline

- Motivation
- Vienna Architecture Description Language (VADL)
- OpenVADL Overview
- Implementation
- Preliminary Results

Motivation

- why another Processor Description Language (PDL) ?
- shortcomings of existing PDLs
  - no separation of concerns (intermingled specifications)
  - often single purpose
  - redundant specifications if multi purpose
  - verbose and error prone
  - slow simulators, compilers, hardware
- hardware processor specifications extremely complex
- compiler specifications extremely complex (GCC, LLVM, GraalVM, V8, ART)
- compilers only support few architectures
- need for a single concise architecture specification for compiler, assembler & linker, (cycle approximate) simulator, hardware, testing, documentation

Vienna Architecture Description Language (VADL)

- VADL is a PDL
- single concise specification without redundancy
- automatic generation of compiler, assembler & linker, simulators and hardware
- syntactic higher order macro system
- tensor and VLIW specifications
- strict separation of Instruction Set Architecture (ISA) and Micro Architecture (MiA)
- MiA specification on a high abstraction level
- Application Binary Interface (ABI) specification
- processor configuration specification

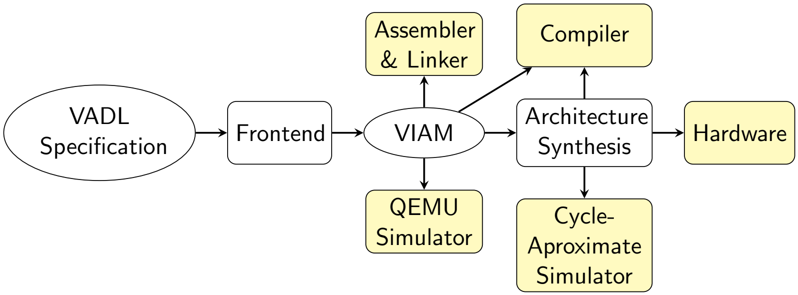
Instruction Set Architecture

```
instruction set architecture RV32I = {

  constant Size = 32          // architecture size is 32 bits

  using Byte    = Bits<  8 > // 8 bit Byte
  using Inst    = Bits< 32 > // instruction word type
  using Regs    = Bits<Size> // register word type
  using Index   = Bits<  5 > // 5 bit register index type for 32 registers
  using Addr    = Regs        // address type is equal to the register type

  [zero : X(0)]               // register with index 0 always is zero
  register X : Index -> Regs // integer register file with 32 registers
  memory MEM : Addr  -> Byte // byte addressed memory
  program counter PC : Addr  // PC points to the start of an instruction

// formats  ...
// models   ...
// model invocations ...

}
```

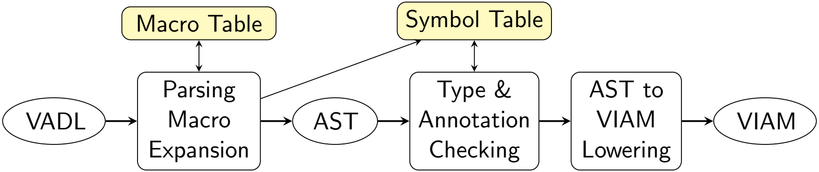
Format, Model and Model Invocation

```
format Itype : Inst =                    // immediate instruction format
  { imm      : Bits<12>                  // [31..20] 12 bit immediate value
  , rs1      : Index                     // [19..15] source register index
  , funct3   : Bits<3>                   // [14..12] 3 bit function code
  , rd       : Index                     // [11..7]  destination register index
  , opcode   : Bits<7>                   // [6..0]   7 bit operation code
  , immS     = imm as SInt<Size>         // sign extended immediate value
  }

// macro for immediate instructions with name, operator, type and opcode
model ImmInstr (name: Id, op: BinOp, type: Id, funct3: Bin) : IsaDefs = {
  instruction $name : Itype =
     X(rd) := (X(rs1) as $type $op immS as $type) as Regs
  encoding $name = {opcode = 0b001'0011, funct3 = $funct3}
  assembly $name =
    (mnemonic," ",register(rd),",",register(rs1),",",decimal(imm))
  }

$ImmInstr (ADDI ; + ; SInt ; 0b000) // add immediate
$ImmInstr (ANDI ; & ; SInt ; 0b111) // and immediate
$ImmInstr (ORI  ; | ; SInt ; 0b110) // or immediate
$ImmInstr (XORI ; ^ ; SInt ; 0b100) // exclusive or immediate
$ImmInstr (SLTI ; < ; SInt ; 0b010) // set less than immediate
$ImmInstr (SLTIU; < ; UInt ; 0b011) // set less than immediate unsigned
```

OpenVADL Overview

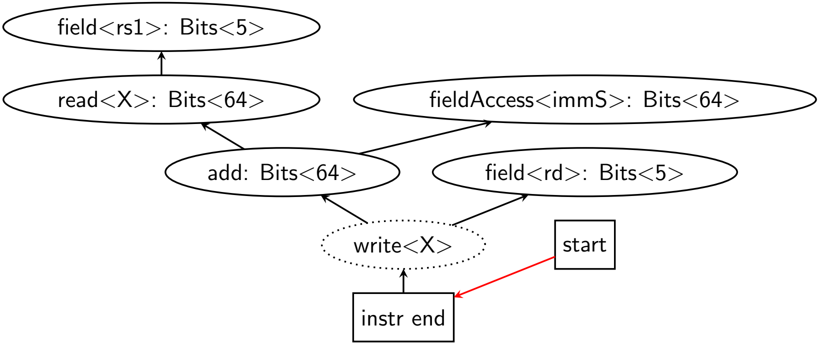


Frontend



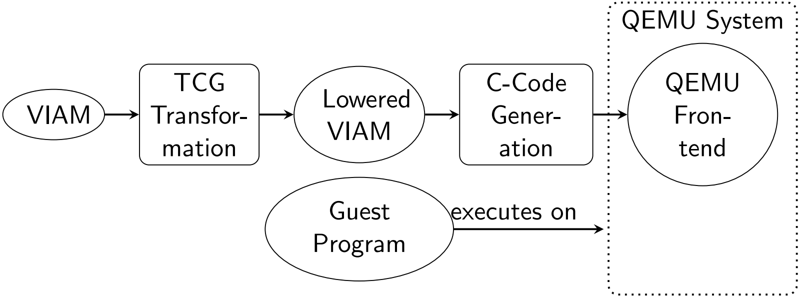
- macros have to be defined before their first use
- constant evaluation on demand
- symbol resolution depends on type checking
- generation of type & annotation checking work in progress
- generation of reference manual planned

VADL Intermediate Architecture Model (VIAM)



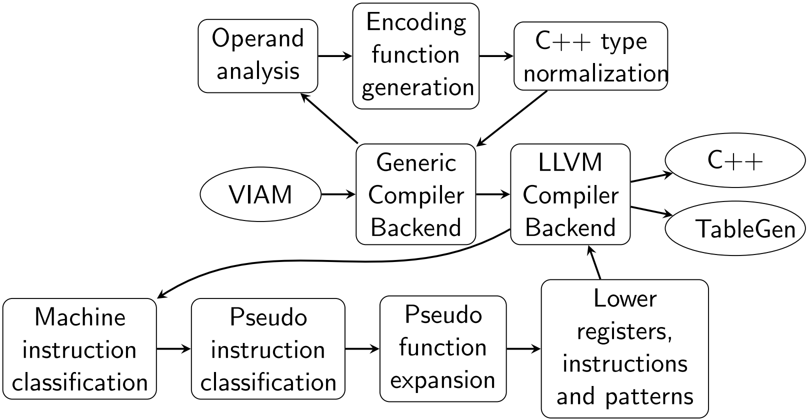
VIAM behavior graph of the RISC-V ADDI instruction

QEMU Frontend Generation

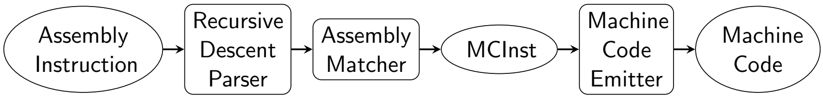


- operation decomposition (mapping to 32/64 bit operation)
- side effect scheduling (register / PC writes)
- scheduling reads before writes
- TCG expression scheduling (translation / run time evaluation)
- branch lowering
- operation lowering

Compiler Generator

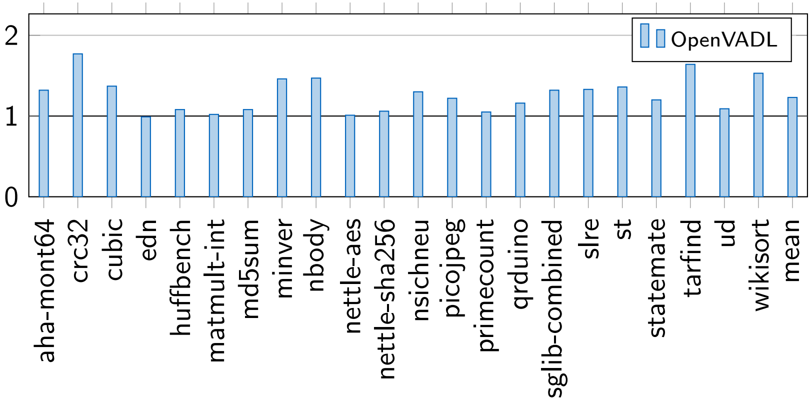


Generated Assembler



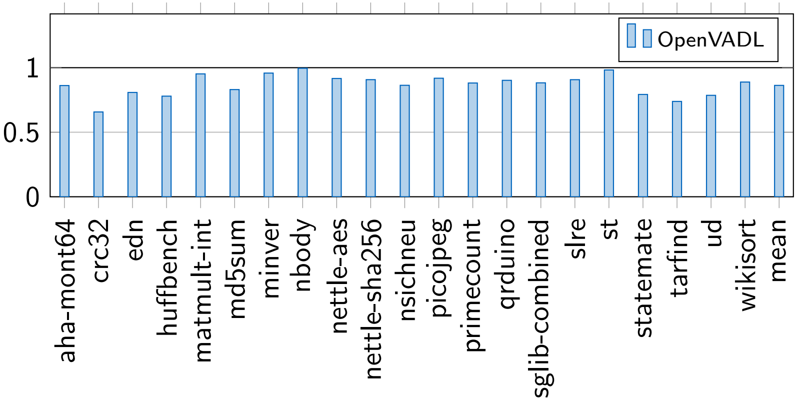
- pred-LL(k) parsing algorithm
- mnemonic / operand vector matched against table of instructions
- LLVM machine code emitter framework

Preliminary Results QEMU Simulator



generated QEMU on average 23% faster than upstream

Preliminary Results LLVM Compiler



generated compiler on average 14% slower than upstream

Conclusion

- enhanced and improved open source implementation of VADL
- architecture synthesis and dependent tools work in progress
- a lot of additional work necessary to improve and complete
- our group is too small for complete development
- we are seeking for cooperation
- `https://openvadl.org`
- `https://github.com/openvadl`

Thank You

Application Binary Interface

```
application binary interface ABI for RV32I = {
  alias register ra = X(1)
  alias register sp = X(2)
//alias register ...

  return address   = ra
  stack  pointer   = sp
  global pointer   = gp
  frame  pointer   = fp
  thread pointer   = tp

  return value      = a{0..1}
  function argument = a{0..7}

  caller saved = [ a{0..7}, t{0..6} ]
  callee saved = [ ra, gp, tp, fp, s{0..11} ]

  constant sequence( rd : Bits<5>, val : SInt<32> ) = {
    LUI  { rd = rd, imm = hi( val ) }
    ADDI { rd = rd, rs1 = rd, imm = lo( val ) }
  }
}
```

Assembly Description

```
directives = {              // rename assembly directives
  ".word" -> BYTE4
  ".quad" -> BYTE8
}

modifiers = {               // modifiers for relocations
  "lo" -> RV32I::lo12,
  "hi" -> RV32I::hi20
}

grammar = {
  ImmediateInstructions @instruction :
    mnemonic = ImmInstrId @operand
    rd  = Register @operand ","
    rs1 = Register @operand ","
    imm = ImmediateOperand ;
  ImmInstrId :
    "ADDI" | "ANDI" | "ORI" | "XORI" | "SLTI" | "SLTIU" ;
}
```

Micro Architecture

```
stage FETCH->(fr:FetchResult)={
    fr := fetchNext
}

stage DECODE->(ir:Instruction)={
  let instr = decode(FETCH.fr) in
  {
    instr.read( @X )
    ir := instr
  }
}

stage EXECUTE->(ir:Instruction)={
  let instr = DECODE.ir in {
    if( instr.unknown ) then
      raise invalid
    else {
      instr.compute
      instr.verify
      instr.write( @PC )
    }
    ir := instr
  }
}
```

Listing 1: Fetch, Decode and Execute Stage

```
stage MEMORY->(ir:Instruction)={
  let instr = EXECUTE.ir in {
    instr.write( @MEM )
    instr.read( @MEM )
    ir := instr
  }
}

stage WRITE_BACK = {
  let instr = MEMORY.ir in {
    instr.write( @X )
  }
}
```

Listing 2: Memory and Write-Back Stage