OpenVADL

An open source implementation of the
Vienna Architecture Description Language

Florian Freitag, Johannes Gisy, Linus Halder
Nathalie Herzog, Benedikt Huber, Benjamin Kasper
Benjamin Komar, Florian Malicky, Michael Nestler
Kevin Per, Matthias Raschhofer, Jakob Rathbauer
Alexander Ripar, Kevin Schlosser, Tobias Weiss
Johannes Zottele, **Andreas Krall**

Compilers and Languages, TU Wien

April 16th 2026

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, Instruction, Encoding and Assembly

```
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
  }

  instruction ADDI : Itype =
     X(rd) := (X(rs1) as SInt + immS as SInt) as Regs
  encoding ADDI = {opcode = 0b001'0011, funct3 = 0b000}
  assembly ADDI =
    (mnemonic," ",register(rd),",",register(rs1),",",decimal(imm))


  instruction SLTIU : Itype =
     X(rd) := (X(rs1) as UInt < immS as UInt) as Regs
  encoding SLTIU = {opcode = 0b001'0011, funct3 = 0b011}
  assembly SLTIU =
    (mnemonic," ",register(rd),",",register(rs1),",",decimal(imm))
```

Model and Model Invocation

```
// 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
```

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 ) }
  }
}
```

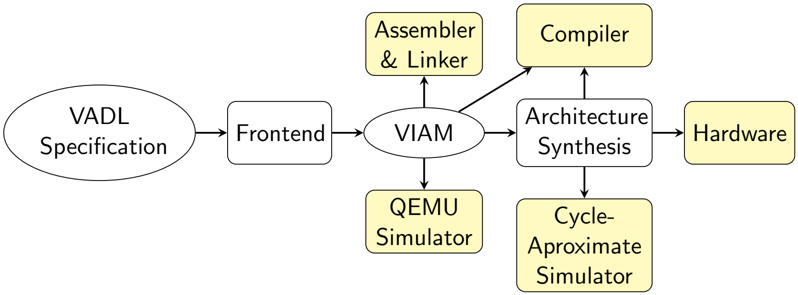
```
stage FETCH->(fr:FetchResult)={
    fr := fetchNext
}

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

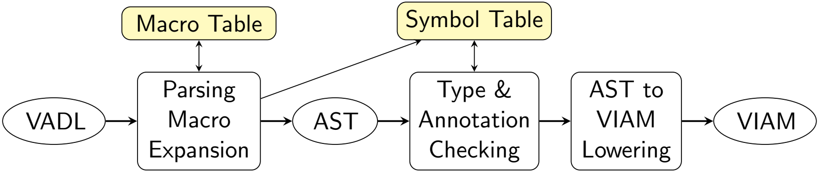
```
stage MEMORY->(ir:Instruction)={
  let instr = EXECUTE.ir in {
    instr.write(@MEM)
    instr.read( @MEM)
    ir := instr
  }
}

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

Listing 2: Memory and Write-Back Stage

OpenVADL Overview

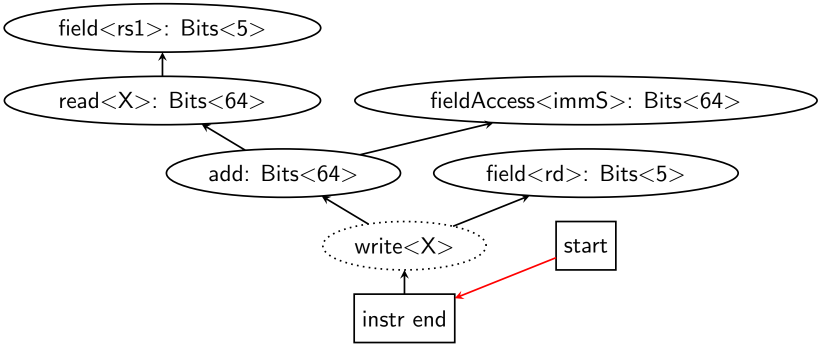


Frontend



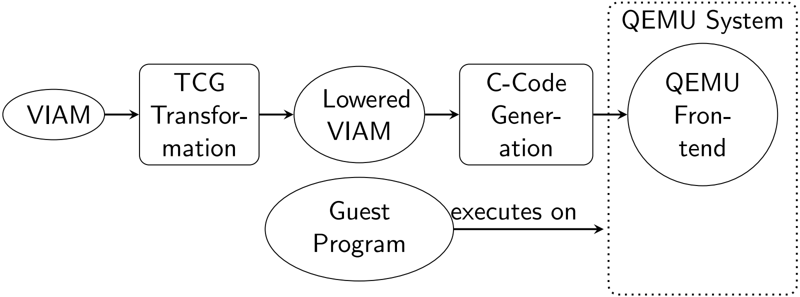
- macros have to be defined before their first use
- constant evaluation on demand
- symbol resolution depends on type checking
- generation of type & annotation checking work in progress
- generation of reference manual planned

VADL Intermediate Architecture Model (VIAM)



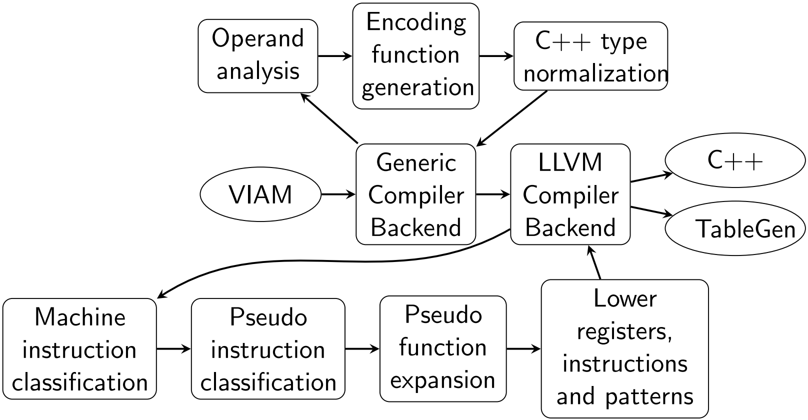
VIAM behavior graph of the RISC-V ADDI instruction

QEMU Frontend Generation

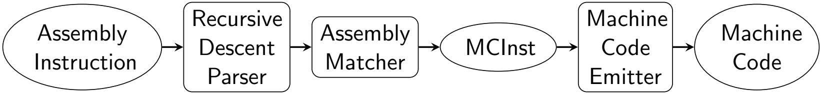


- operation decomposition (mapping to 32/64 bit operation)
- side effect scheduling (register / PC writes)
- scheduling reads before writes
- TCG expression scheduling (translation / run time evaluation)
- branch lowering
- operation lowering

Compiler Generator



Generated Assembler



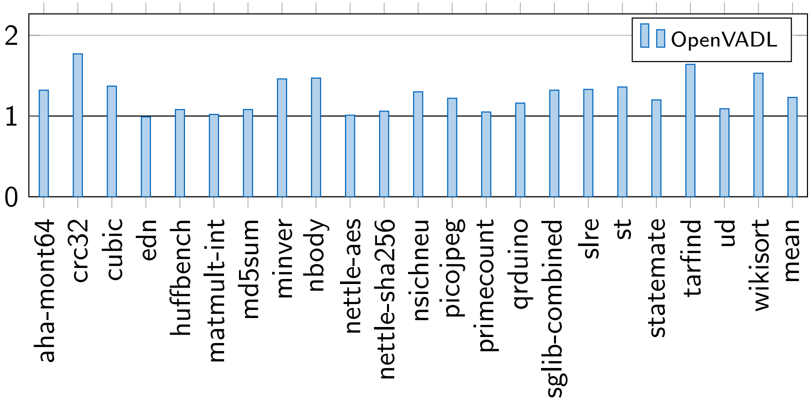
- pred-LL(k) parsing algorithm
- mnemonic / operand vector matched against table of instructions
- LLVM machine code emitter framework

Architecture Synthesis

```
instruction ADDI : Itype = {
  X(rd) := X(rs1) + immS
}


instruction SW : Stype = {
  let addr = X(rs1) + immS in

  let res  = X(rs2) in
    MEM<4>(addr) := res as Word
}
```

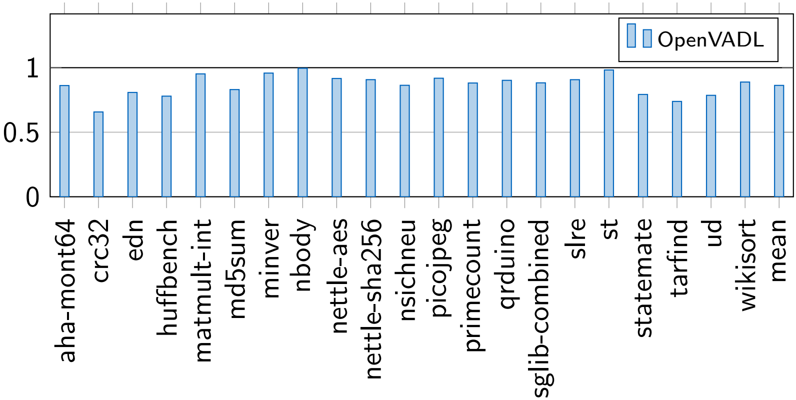
```
stage DECODE->(ir : Instruction)={
  let instr=decode(FETCH.fr) in {
    instr.read(@X)
    ir := instr
  }
}

stage EXECUTE->(ir : Instruction)={
  let instr=DECODE.ir in
    instr.compute
}
```

Preliminary Results QEMU Simulator (RISC-V RV64IM)



generated QEMU on average 23% faster than upstream

Preliminary Results LLVM Compiler (RISC-V RV32IM)



generated compiler on average 14% slower than upstream

Preliminary Results Hardware Generation (RISC-V RV32I)

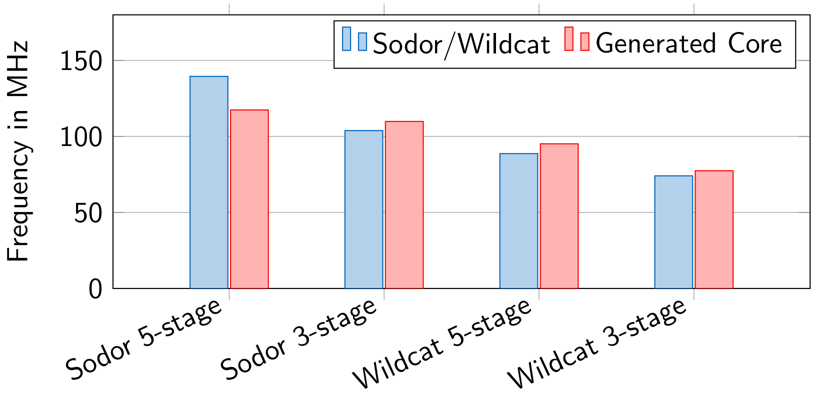


Figure: Maximum clock frequency of synthesized RV32I cores with the SkyWater130 ASIC process (higher is better).

Conclusion

- enhanced and improved open source implementation of VADL
- a lot of additional work necessary to improve and complete
- many theses necessary for complete development
- start of theses possible at any time
- `https://openvadl.org`
- `https://github.com/openvadl`

Thank You