



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  
Benjamin Komar, Jakob Rathbauer, Martin Resch, Kevin  
Schlosser, **Andreas Krall**

Compilers and Languages, TU Wien

July 22nd 2025

- why another Processor Description Language (PDL) ?
- shortcomings of existing PDLs
- hardware processor specifications extremely complex
- compiler implementations extremely complex (GCC, LLVM, ART, GraalVM, V8)
- need for a single concise architecture specification for compiler, assembler & linker, (cycle approximate) simulator, hardware, testing, documentation
- no redundancy in specification parts
- specification is completely tested on simulator (co-simulation)
- correct specification is input for other generators

# 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

```
1 instruction set architecture RV32I = {
2
3   constant Size = 32           // architecture size is 32 bits
4
5   using Byte    = Bits< 8 > // 8 bit Byte
6   using Inst    = Bits< 32 > // instruction word type
7   using Regs    = Bits<Size> // register word type
8   using Index   = Bits< 5 > // 5 bit register index type for 32 registers
9   using Addr    = Regs       // address type is equal to the register type
10
11   [zero : X(0)]                // register with index 0 always is zero
12   register X : Index → Regs    // integer register file with 32 registers
13   memory MEM : Addr → Byte    // byte addressed memory
14   program counter PC : Addr // PC points to the start of an instruction
15
16   // formats ...
17   // models ...
18   // model invocations ...
19
20 }
```

# Format, Instruction, Encoding and Assembly

```
1 format Itype : Inst = // immediate instruction format
2 { imm : Bits<12> // [31..20] 12 bit immediate value
3 , rs1 : Index // [19..15] source register index
4 , funct3 : Bits<3> // [14..12] 3 bit function code
5 , rd : Index // [11..7] destination register index
6 , opcode : Bits<7> // [6..0] 7 bit operation code
7 , immS = imm as SInt<Size> // sign extended immediate value
8 }
9
10 instruction ADDI : Itype =
11 X(rd) := (X(rs1) as SInt + immS as SInt) as Regs
12 encoding ADDI = {opcode = 0b001'0011, funct3 = 0b000}
13 assembly ADDI =
14 (mnemonic, " ", register(rd), ", ", register(rs1), ", ", decimal(imm))
15
16
17 instruction SLTIU : Itype =
18 X(rd) := (X(rs1) as UInt < immS as UInt) as Regs
19 encoding SLTIU = {opcode = 0b001'0011, funct3 = 0b011}
20 assembly SLTIU =
21 (mnemonic, " ", register(rd), ", ", register(rs1), ", ", decimal(imm))
```

# Model and Model Invocation

```
1 // macro for immediate instructions with name, operator, type and opcode
2 model Immlnstr (name: Id, op: BinOp, type: Id, funct3: Bin) : IsaDefs = {
3   instruction $name : ltype =
4     X(rd) := (X(rs1) as $type $op immS as $type) as Regs
5   encoding $name = {opcode = 0b001'0011, funct3 = $funct3}
6   assembly $name =
7     (mnemonic, " ", register(rd), ", ", register(rs1), ", ", decimal(imm))
8   }
9
10 $Immlnstr (ADDI ; + ; SInt ; 0b000) // add immediate
11 $Immlnstr (ANDI ; & ; SInt ; 0b111) // and immediate
12 $Immlnstr (ORI ; | ; SInt ; 0b110) // or immediate
13 $Immlnstr (XORI ; ^ ; SInt ; 0b100) // exclusive or immediate
14 $Immlnstr (SLTI ; < ; SInt ; 0b010) // set less than immediate
15 $Immlnstr (SLTIU ; < ; UInt ; 0b011) // set less than immediate unsigned
```

# Application Binary Interface

```
1 application binary interface ABI for RV32I = {
2   alias register ra = X(1)
3   alias register sp = X(2)
4   //alias register ...
5
6   return address    = ra
7   stack pointer    = sp
8   global pointer   = gp
9   frame pointer    = fp
10  thread pointer    = tp
11
12  return value      = a{0..1}
13  function argument = a{0..7}
14
15  caller saved = [ a{0..7}, t{0..6} ]
16  callee saved = [ ra, gp, tp, fp, s{0..11} ]
17
18  constant sequence( rd : Bits<5>, val : SInt<32> ) = {
19    LUI { rd = rd, imm = hi( val ) }
20    ADDI { rd = rd, rs1 = rd, imm = lo( val ) }
21  }
22 }
```

# Assembler Parser Description

```
1 directives = {           // rename assembly directives
2   ".word" → BYTE4
3   ".quad" → BYTE8
4 }
5
6 modifiers = {           // modifiers for relocations
7   "lo" → RV32I::lo12,
8   "hi" → RV32I::hi20
9 }
10
11 grammar = {
12   ImmediateInstructions @instruction :
13     mnemonic = Immlnstrld @operand
14     rd = Register @operand ","
15     rs1 = Register @operand ","
16     imm = ImmediateOperand ;
17   Immlnstrld :
18     "ADDI" | "ANDI" | "ORI" | "XORI" | "SLTI" | "SLTIU" ;
19 }
```

```
1 stage FETCH->(fr:FetchResult)={
2   fr := fetchNext
3 }
4
5 stage DECODE->(ir:Instruction)={
6   let instr = decode(FETCH.fr) in
7   {
8     instr.read(@X)
9     ir := instr
10  }
11 }
12
13 stage EXECUTE->(ir:Instruction)={
14   let instr = DECODE.ir in {
15     if (instr.unknown) then
16       raise invalid
17     else {
18       instr.compute
19       instr.verify
20       instr.write(@PC)
21     }
22     ir := instr
23   }
24 }
```

Listing 1: Fetch, Decode and Execute Stage

```
1 stage MEMORY->(ir:Instruction)={
2   let instr = EXECUTE.ir in {
3     instr.write(@MEM)
4     instr.read(@MEM)
5     ir := instr
6   }
7 }
8
9 stage WRITE.BACK = {
10  let instr = MEMORY.ir in {
11    instr.write(@X)
12  }
13 }
```

Listing 2: Memory and Write-Back Stage

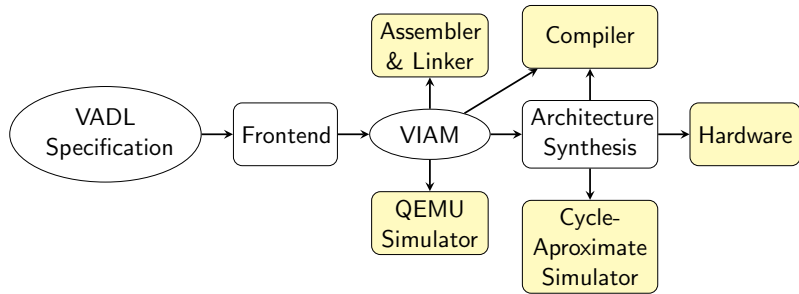
```
1 logic [forwarding] bypass
2
3 stage DECODE->(ir:Instruction)={
4   let instr = decode(FETCH.fr) in {
5     instr.readOrForward(@X,@bypass)
6     ir := instr
7   }
8 }
9
10 stage EXECUTE->(ir:Instruction)={
11   let instr = DECODE.ir in {
12     if( instr.unknown ) then
13       raise invalid
14     else {
15       instr.compute
16       instr.results(@bypass)
17       instr.verify
18       instr.write(@PC)
19     }
20     ir := instr
21   }
22 }
```

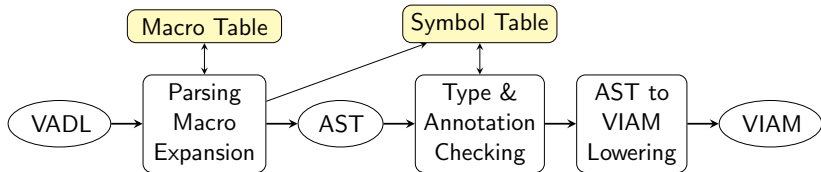
Listing 3: Forwarding Logic

```
1 [ write through      ]
2 [ evict roundrobin  ]
3 [ entries = 1024    ]
4 [ blocks = 4        ]
5 [ n_set = 2         ]
6 [ attached_to MEM   ]
7 cache L1: Addr -> Byte
```

Listing 4: Cache Definition

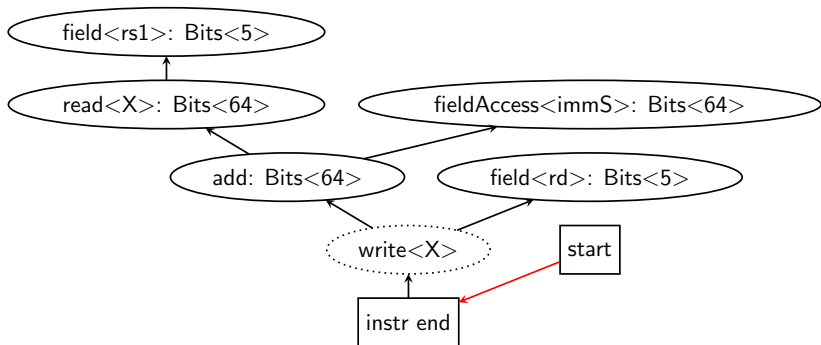
# OpenVADL Overview





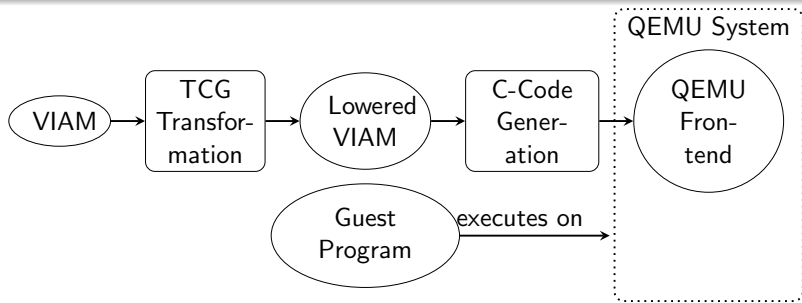
- macros have to be defined before their first use
- constant evaluation on demand
- symbol resolution depends on type checking
- generation of type & annotation checking work in progress
- generation of reference manual planned

# VADL Intermediate Architecture Model (VIAM)



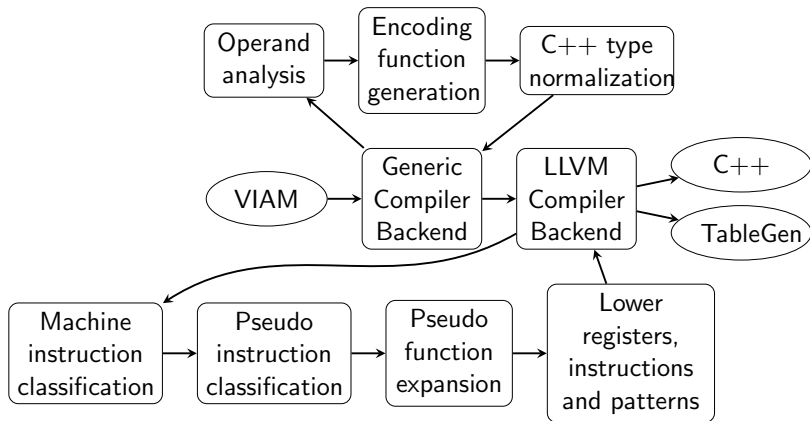
VIAM behavior graph of the RISC-V ADDI instruction

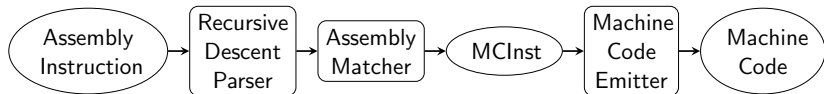
# QEMU Frontend Generation



- 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





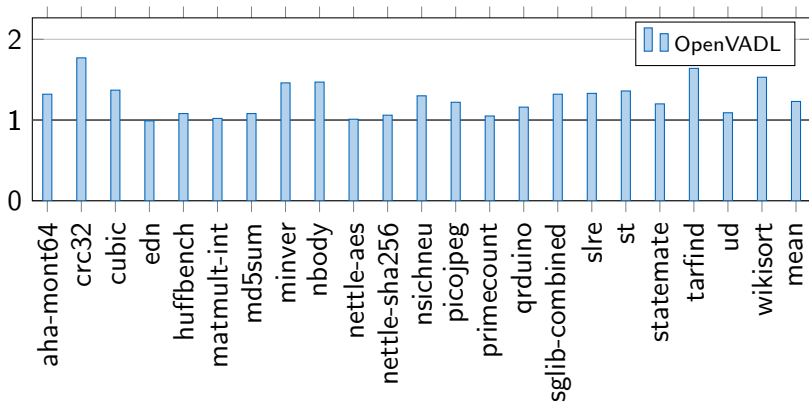
- pred-LL(k) parsing algorithm
- mnemonic / operand vector matched against table of instructions
- LLVM machine code emitter framework

# Architecture Synthesis

```
1 instruction ADDI : ltype = {  
2   X(rd) := X(rs1) + immS  
3 }  
4  
5  
6 instruction SW : Stype = {  
7   let addr = X(rs1) + immS in  
8   let res = X(rs2) in  
9   MEM<4>(addr) := res as Word  
10 }  
11 }
```

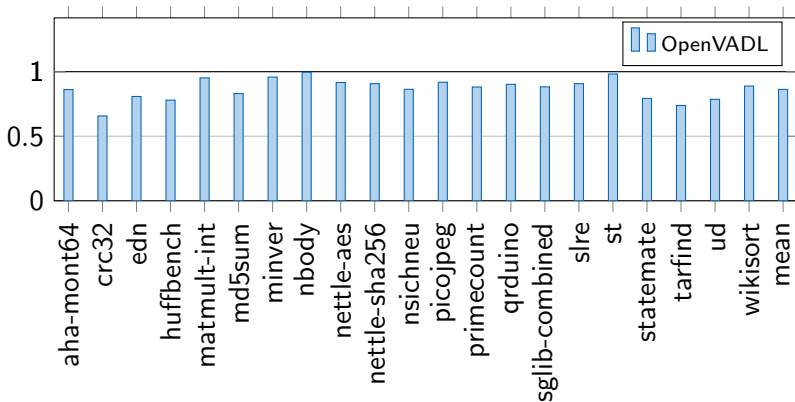
```
1 stage DECODE->(ir: Instruction)={  
2   let instr=decode(FETCH.fr) in {  
3     instr.read(@X)  
4     ir := instr  
5   }  
6 }  
7  
8 stage EXECUTE->(ir: Instruction)={  
9   let instr=DECODE.ir in  
10  instr.compute  
11 }
```

# 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

- 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**