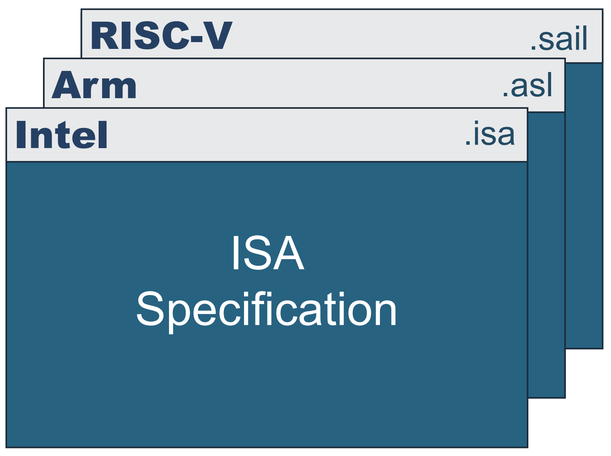
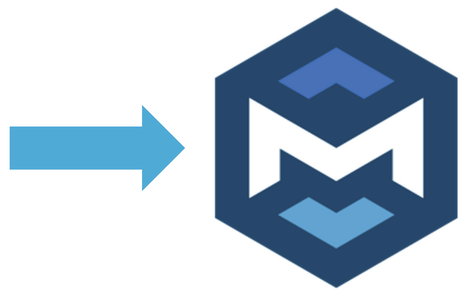
Connecting MLIR with ISA specifications (WIP)

Alastair Reid
Intel Inc.

https://alastairreid.github.io/papers/#talks



Trustworthy, complete, authoritative, machine-readable architecture specifications



Readable, trustworthy documentation

- Compilers
  - Compiler generators (i.e., Instruction selection)
  - Discovery, verification and synthesis of peephole optimizations
  - Translation validation
  - Generating JITs
  - Verifying compilers
- Formally verifying software
  - Hand-written cryptography code
  - OS code (e.g., page table modification, etc.)
- Formally verifying processors
  - Instruction pipelines
  - Microcode
- Architecture Design
  - Generate simulators
  - Testing architecture design
  - Automatic generation of test cases
  - Verifying architecture design invariants
- …

See https://alastairreid.github.io/uses-for-isa-specs/

Challenge #1: Compiler view vs. ISA view

MLIR operations	ISA instructions
SSA inputs and outputs (infinite # registers)	32 registers specified by 5-bit operand
No side effects	Memory address modes
No flags	Side effects on flags, IP/PC, …
Branch targets unconstrained	Implementation defined behavior
	Privilege checks, CPUID checks, …
	Conditional branch targets must be +/- 128 bytes from branch

What is the formal relationship between an MLIR instruction and a machine instruction?
Can we automatically generate wrappers to convert between different views?

Challenge #2: Non-standard dialects and operations

Multiprecision integer type
Sized bitvector type “Bits(96)”
Bitslice operations: “result[i +: 32]”
Semi-structured control flow (not SCF’s “single-entry, single-exit)
- Early return from function (e.g., inside if-statement or loop): “`if size == 0 then return; endif`”
- Throwing exceptions: “`throw Exception_Taken;`”
- Propagating exceptions: “`Check_Read_Permission?(segment);`”
- Pattern-matching case statement

```
“case x of
    when 0b0000 => return False;
    when 0b1xxx if mode == Mode::Protected => Undefined!();
    when 0b1xxx => return True;
    otherwise => Reserved!();
 endcase”
```

Wait for dialects to be standardized?
Use dialects proposed for standardization and hope proposal is accepted?
Lower to standard dialect – but lose useful semantic information?

Executable Instruction specifications

```
⟪Operation (vector integer addition)⟫:                    // One of 700+ instruction families
    let elements := register_size / element_size;
    var result := Zero(register_size);
    for i := 0 to elements-1 do
        let op1 := src1[i *: element_size];  // read ith element
        let op2 := src2[i *: element_size];
        let r := op1 + op2;
        result[i *: element_size] := r;      // write ith element
    endfor;

Assembly syntax and encodings:
    VPADDW ymm1, ymm2, m256          VPADDW_YMMqq_YMMqq_MEMqq     VEX.L0.66.0F.WIG FD /r
    VPADDD ymm1, ymm2, m256          VPADDD_YMMqq_YMMqq_MEMqq     VEX.L0.66.0F.WIG FE /r
    … 50+ other intrinsics omitted …

⟪VPADDD_YMMqq_YMMqq_MEMqq operand read/write⟫:
    let register_size := 256;
    let element_size := 32;
    let disp8n := 1;
    let (ea_offset, segment, next_ip0, is_rip_relative):= Effective_Address?(address_size, context, mod, rm, disp8n, ip);
    let src1 := Read_YMM(context.evex_v4 ++ context.vvvv);
    let effective_address := Handle_RIP_Relative_Address(address_size, is_rip_relative, ea_offset, next_ip0);
    let src2 := Logical_Mem_Read?(segment, effective_address, register_size);
    let next_ip := ip;
    let result := ⟪Operation (vector integer addition)⟫;
    Write_YMM(context.rex_r4 ++ context.rex_r3 ++ reg, result);

⟪Intrinsic⟫:                                              // 4 of the 35 intrinsics for this instruction family
    __m64 _m_add_epi32(__m64 a, __m64 b);                 // PADDW_MMXq_MMXq
    __m128i _mm_add_epi32(__m128i a, __m128i b);          // PADDW_XMMdq_XMMdq
    __m256i _mm256_add_epi32(__m256i a, __m256i b);       // VPADDB_YMMqq_YMMqq_YMMqq
    __m512i _mm512_add_epi32(__m512i a, __m512i b);       // VPADDW_ZMMu16_MASKmskw_ZMMu16_ZMMu16_AVX512
```

Helper functions

```
// One of 900+ helper functions
function Logical_Mem_Read?(
      segment : Segment, offset : Bits(64), size : {8, 32, 64, 128, 256, 512},
      access_type : Memory_Access_Type := Memory_Read,
      alignment_type : Memory_Alignment_Type := Normal_Alignment,
      acquire_lock : Boolean := False
   ) -> Bits(size)
begin
   Check_Read_Permission?(segment);

   if access_type == Memory_Read_Modify_Write then
      Check_Write_Permission?(segment);
   endif;

   let logical := Create_Logical_Address(segment, offset, size);
   let linear := Translate_Logical_To_Linear?(logical, alignment_type);
   Check_Data_Read_Breakpoint(linear.linear_address, size / 8);

   let physical := Translate_Linear_To_Physical?(linear, access_type);
   …
   let value := Physical_Mem_Read(physical, size);
   return value;
end
```

Challenge #3: Rich type systems

Constrained types: “`size : {8, 32, 64, 128, 256, 512}`”
Dependent types:

“`function Std::Bits::Reverse(implicit size : {0..}, x : Bits(size)) -> Bits(size)`”
(Roughly: “function Reverse : ∀ size. Bits(size) -> Bits(size)”
or “`function Reverse(size: Integer, x : Bits) -> (r: Bits)`
`  requires size >= 0 and Length(x) == size;`
`  ensures Length(r) == size;`”)

Flow sensitive type (only in SAIL)

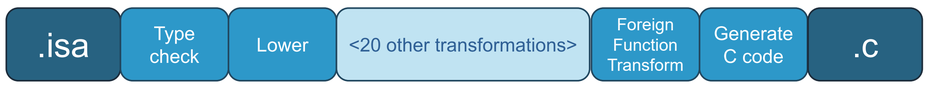
```
if size == 64 then
     Zeros(64);          // has type Bits(64)
 else
     All_Ones(size);     // has type Bits(size)
endif
```

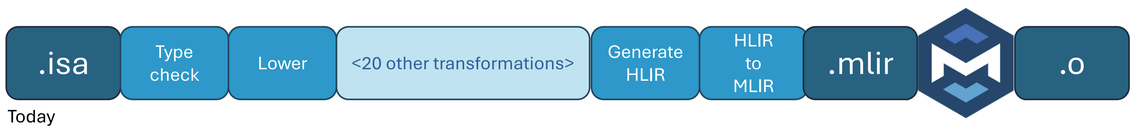
Embed richer type system in MLIR attributes?
Replace static typechecks with dynamic (runtime) checks?

Non-technical challenges

- *Scale*: Around 100-150 built in operations in .isa/.asl/.sail to support
- *The cost of success*: Supporting existing users while performing heart+lung transplant on toolchain
- *Chicken and egg*: successful use-cases justify investing effort in (significant) re-engineering effort
- Limited developer bandwidth

Evolving the current toolchain





Today

… time passes …

https://github.com/IntelLabs/isa-tools/

Related work (very abbreviated)

Formal ISA specifications
Falkoff++, A formal description of SYSTEM/360, IBM Systems Journal, 1964
Bell & Newell, The PMS and ISP descriptive systems for computer structures, AFIPS 1970
Fox & Myreen, A trustworthy monadic formalization of the ARMv7 instruction set architecture, ITP 2010
Degenbaev, Formal specification of the x86 instruction set architecture, PhD 2012
Heule+, Stratified synthesis: Automatically learning the x86-64 instruction set, PLDI 2016
Reid, Trustworthy specifications of ARM v8-A and v8-M system level architecture, FMCAD 2016
Armstrong++, ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS, POPL 2019
Dasgupta, A complete formal semantics of x86-64 user-level instruction set architecture, PLDI 2019
Roessle, Formally verified big step semantics out of x86-64 binaries, CPP 2019

Decompilation
Dasgupta, Scalable validation of binary lifters, PLDI 2020

Discovering and verifying peephole optimizations
Davidson+, Automatic Generation of Peephole Optimizations, CC 1984
Bansal+, Automatic generation of peephole superoptimizers, ASPLOS 2006
Bansal+, Binary translation using peephole superoptimizers, OSDI 2008
Mukherjee+, Hydra: Generalizing Peephole Optimizations with Program Synthesis, OOPSLA 2024

Generating testsuites
Martignoni+, Path-exploration lifting: Hi-fi tests for lo-fi emulators, ASPLOS 2012

Verifying compilers
Samet, Automatically proving the correctness of translations involving optimized code, PhD 1975
Leroy, Formal verification of a realistic compiler, CACM 2009
Fox+, Verified compilation of CakeML to multiple machine-code targets, CPP 2017

Verifying processors
Reid+, End-to-end verification of ARM processors with ISA-formal
Goel, Using x86isa for microcode verification, SpISA 2019
Goel, Verifying x86 instruction implementations, CPP 2019

See https://alastairreid.github.io/RelatedWork/notes/isa-specification/