



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

How can you use ISA specifications in your compiler?

Alastair Reid (Intel)

26 October 2026

[R278](#)

Cambridge University

Overview

- What compilers need to know about instructions
- What is an ISA specification and how to use them?
 - Traditional ISA specs
 - The (ongoing) revolution in ISA specs
- Converting ISA specs to MLIR
- How to prove that VeIR ISA specs refine official ISA specs?
- How to generate VeIR ISA specs from official ISA specs?

Note: some parts of this story have holes – that I'm hoping some of you will fill!

Changing compiler ecosystems

Compilers support many different architectures

Targetting just one architecture

- Demonstrates feasibility
- Solves (maybe?) 50% of the challenges found across all architectures
- Doesn't motivate ecosystem change

My focus

- Solutions that can scale to the most popular/influential/important/representative architectures
- Intel x86, Arm, RISC-V

What compilers know about instructions

1. Encoding

- “vpaddq %ymm0, %ymm1, %ymm2” → “c5 f5 d4 c2”

2. Instruction selection

- “%r = add <4 x i64> %x, %y” → “vpaddq %ymm0, %ymm1, %ymm2”

3. Optimization

- “vpaddq(vpmadd52(<zero>, a, b), x)” → “vpmadd52(x, a, b)”

4. Does instruction have side effects? Throw exceptions? ...

- See “Veir/Dialects/RISCV_Cf/OpInfo.lean”

How would we prove that each of these is correct?

The old way (x86)

VPADDD (VEX.256 Encoded Instruction)

`DEST[31:0] := SRC1[31:0] + SRC2[31:0];`

`(* Repeat add operation for 2nd and 7th doubleword *)`

`DEST[255:224] := SRC1[255:224] + SRC2[255:224];`

→ {manual transcription}

Lean/Rocq/HOL/SMT/...

The old way (Arm v7)

VADD<c>.<dt> <Qd>, <Qn>, <Qm>

```
if ConditionPassed() then
  EncodingSpecificOperations();
  CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      Elem[D[d+r],e,esize]
        = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

→ {manual transcription}

Lean/Rocq/HOL/SMT/...

Problems

- Inaccuracies in original spec (not tested/testable)
- Inaccuracies in transcription
- Limited testing of Lean/Rocq/... specification
- Specification only usable within one theorem prover
- Specification omits ‘unimportant’ details
 - Would they matter if you want to use spec for something else?
- Is specification updated as architecture changes?

Ongoing revolution in ISA specifications

ARM Releases Machine Readable Architecture

Specification *April 20, 2017.*

Today ARM released version 8.2 of the ARM v8-A processor specification in machine readable form. This specification describes almost all of the architecture: instruction page table walks, taking interrupts, taking synchronous exceptions such as page faults, taking asynchronous exceptions such as bus faults, user mode, system mode, hypervisor mode, secure mode, debug mode. It details all the instruction formats and system registers. The semantics is written in [ARM's ASL Specification Language](#) so it is all machine executable and has been tested very thoroughly using the same architecture conformance tests that ARM uses to test its processors. (See my paper "[Trustworthy Specification of ARM v8-A and v8-M System Level Architecture](#)".)

An Introduction to the Official Formal Specification of the RISC-V Instruction Set Architecture

Rishiyur S. Nikhil, PhD ¹
CTO and co-founder, Bluespec, Inc.
(nikhil@bluespec.com)

3rd RISC-V Meeting, March 30-31, 2021, France

A preview of the future IA documentation

Aug 20, 2026

The behavior of instructions and helper functions is described in a [new specification language](#) that enables us to test the specification which makes the specification more precise and accurate.

Modern ISA Specifications

	Intel x86	Intel GPU	RISC-V	Arm AArch32	Arm AArch64	Arm M-class
Executable & Tested	Yes	Yes	Yes	Yes	Yes	Yes
Machine-readable spec available	Yes	No	Yes	Yes	Yes	?
In official (public) documentation	Soon	No	Partially integrated	Yes	Yes	Yes
Formally verified w/ processor	Eventually	Yes	Yes	Yes	Yes	Yes
Tools available	Yes	Yes	Yes	Partial	Partial	Partial
Boots Linux	?	N/A	Yes	Yes	Yes	N/A
Language	.isa	.isa	.sail	.asl	.asl	.asl

The new way (x86)

```
1  let elements := register_size / element_size;
2  var result := Zero(register_size);
3  for i := 0 to elements-1 do
4      let op1 := src1[i *: element_size];
5      let op2 := src2[i *: element_size];
6      let r := op1 + op2;
7      result[i *: element_size] := r;
8  endfor;
```

→ {automatically}

C / LLVM-IR / MLIR

The new way (RISC-V)

```
function clause execute RTYPE(rs2, rs1, rd, op) = {  
  X(rd) = match op {  
    ADD => X(rs1) + X(rs2),  
    SLT => zero_extend(bool_to_bit(X(rs1) <_s X(rs2))),  
    ...  
  }
```

→ {automatically}

.c / .sv / Rocq / .smt / <your verification language here?>

Generating MLIR

riscv.isa (829 lines) → riscv.mlir (9955 lines)

```
git clone https://github.com/alastairreid/isa-tools.git -b areid-mlir
cd isa-tools
make
./test.sh
less t1.mlir
```

Beware: incomplete - some .isa standard library operations currently unsupported

```

function Instr::ADD
  (rs2 : Bits(5), rs1 : Bits(5), rd : Bits(5))
begin
  let s1 := Unsigned(rs1);
  let s2 := Unsigned(rs2);
  let d := Unsigned(rd);
  let operand1 := X[s1];
  let operand2 := X[s2];
  let result := operand1 + operand2;
  X[d] := result;
end

```



```

iii --exec=':to_mlir --no-typecheck'
mlir-opt --inline --canonicalize

```

```

func.func @Instr$ADD(%arg0: i128, %arg1: i128, %arg2: i128) {
  %c0_i64 = arith.constant 0 : i64
  %c32_i64 = arith.constant 32 : i64
  %0 = arith.trunci %arg1 : i128 to i63
  %1 = arith.extui %0 : i63 to i64
  %2 = arith.trunci %arg0 : i128 to i63
  %3 = arith.extui %2 : i63 to i64
  %4 = arith.trunci %arg2 : i128 to i63
  %5 = arith.extui %4 : i63 to i64
  %6 = arith.cmpi eq, %1, %c0_i64 : i64
  cf.cond_br %6, ^bb1, ^bb2
^bb1: // pred: ^bb0
  %7 = call @Std$Bits$Zero(%c32_i64) : (i64) -> i128
  cf.br ^bb3(%7 : i128)
^bb2: // pred: ^bb0
  %8 = index.casts %1 : i64 to index
  %9 = memref.get_global @_X : memref<32xi128>
  %10 = memref.load %9[%8] : memref<32xi128>
  cf.br ^bb3(%10 : i128)
^bb3(%11: i128): // 2 preds: ^bb1, ^bb2
  %12 = arith.cmpi eq, %3, %c0_i64 : i64
  cf.cond_br %12, ^bb4, ^bb5
^bb4: // pred: ^bb3
  %13 = call @Std$Bits$Zero(%c32_i64) : (i64) -> i128
  cf.br ^bb6(%13 : i128)
^bb5: // pred: ^bb3
  %14 = index.casts %3 : i64 to index
  %15 = memref.get_global @_X : memref<32xi128>
  %16 = memref.load %15[%14] : memref<32xi128>
  cf.br ^bb6(%16 : i128)
^bb6(%17: i128): // 2 preds: ^bb4, ^bb5
  %18 = arith.addi %11, %17 : i128
  %19 = arith.cmpi ne, %5, %c0_i64 : i64
  cf.cond_br %19, ^bb7, ^bb8
^bb7: // pred: ^bb6
  %20 = index.casts %5 : i64 to index
  %21 = memref.get_global @_X : memref<32xi128>
  memref.store %18, %21[%20] : memref<32xi128>
  cf.br ^bb8
^bb8: // 2 preds: ^bb6, ^bb7
  return
}

```

What does VeIR's spec look like?

Veir/Data/RISCV/Reg/Basic.lean:

```
--  
  Adds the registers rs1 and rs2 and stores the result in rd.  
  Arithmetic overflow is ignored and the result is simply the  
  low 64 bits of the result.  
-/  
def add (rs2 : Reg) (rs1 : Reg) : Reg :=  
  ⟨rs1.val + rs2.val⟩  
  
structure Reg where  
  val : BitVec 64  
deriving DecidableEq
```

Is VeIR's spec equivalent to the official spec?

function clause execute

RTYPE(rs2, rs1, rd, op)

= {

X(rd) = match op {

ADD => X(rs1) + X(rs2),

SLT => ...,

SLTU => ...,

...

}

}

def add (rs2 : Reg)

(rs1 : Reg)

: Reg

:= ⟨rs1.val + rs2.val⟩

Is VeIR's spec equivalent to the official spec?

```
function execute_RTYPE_ADD      def add (rs2 : Reg)
    (rs2 : bits(5),              (rs1 : Reg)
    rs1 : bits(5),               : Reg
    rd : bits(5))               := ⟨rs1.val + rs2.val⟩

=

{
  X(rd) = X(rs1) + X(rs2)
}
```

The semantic gap

ISA specifications

- Stateful
- Register indexes
- Exceptions
 - Divide by 0 / Page fault, ...
 - Well defined
 - Catchable (in OS kernel)

VeIR / xDSL / MLIR operations

- Pure
- SSA values
- Exceptions
 - Poison / Undefined / ...
 - Ill-defined
 - Fatal (“OS” is not part of model)

Solution? Intel Intrinsics Guide

```
__m256i _mm256_add_epi64 (__m256i a, __m256i b)
```

Synopsis

```
__m256i _mm256_add_epi64 (__m256i a, __m256i b)
```

```
#include <immintrin.h>
```

```
Instruction: vpaddq ymm, ymm, ymm
```

```
CPUID Flags: AVX2
```

Description

Add packed 64-bit integers in `a` and `b`, and store the results in `dst`.

Operation

```
FOR j := 0 to 3
    i := j*64
    dst[i+63:i] := a[i+63:i] + b[i+63:i]
ENDFOR
dst[MAX:256] := 0
```

<https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=vpaddq>

How to handle the semantic gap?

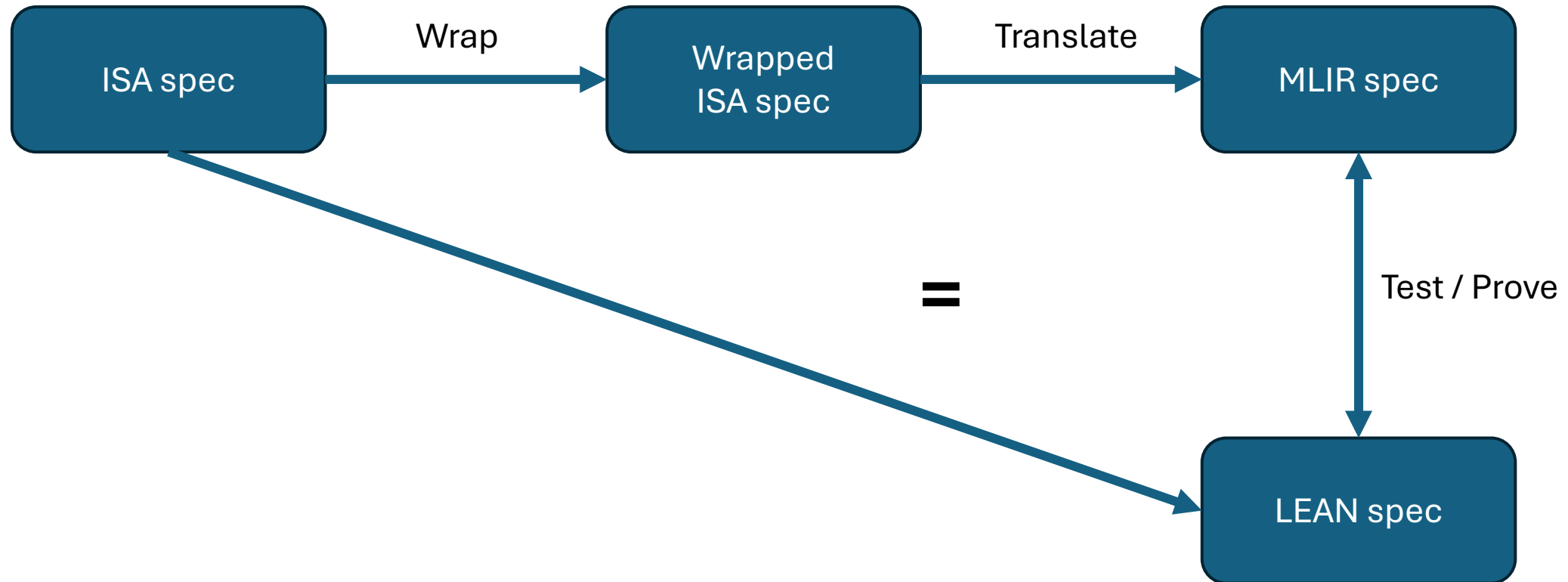
1. Ignore the official ISA specification
2. Manually compare against the official ISA specification
3. Test Lean spec against the official ISA specification
4. Formally verify Lean spec against the official ISA specification
5. Automatically generate Lean from the official ISA specification

How to test / formally verify: Pure wrapper

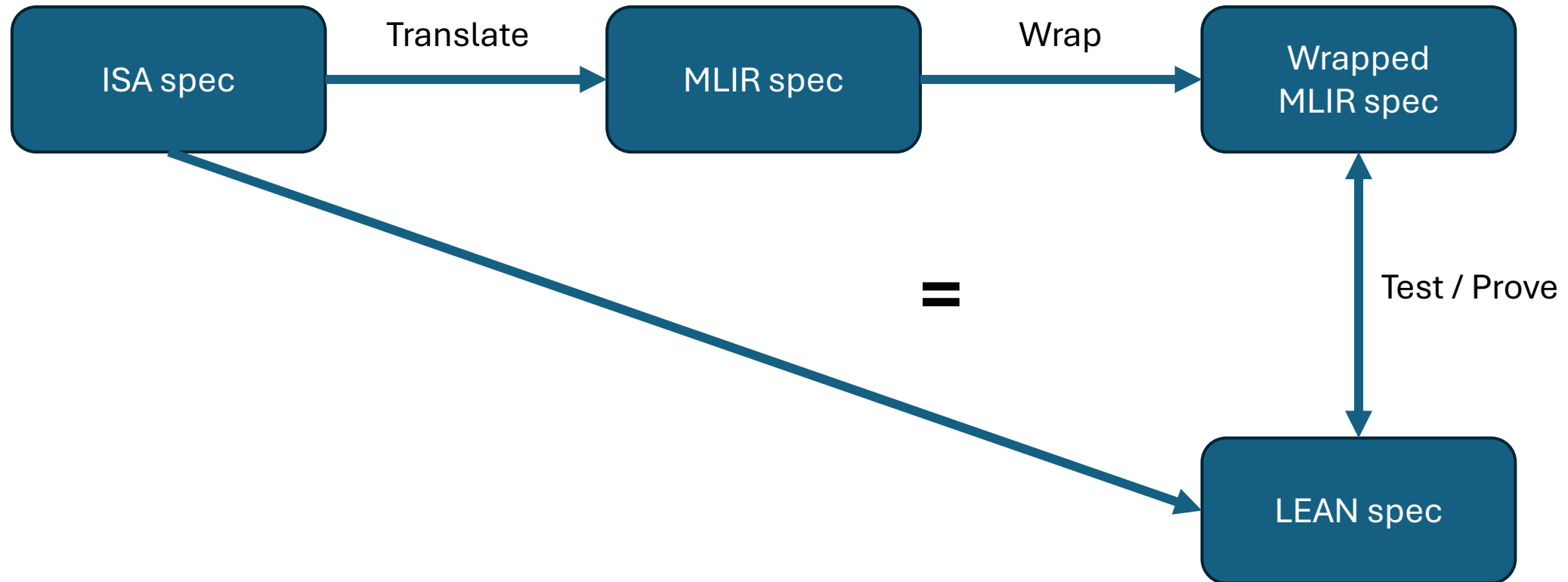
```
function clause execute
  RTYPE(rs2, rs1, rd, op)
= {
  X(rd) = match op {
    ADD => X(rs1) + X(rs2),
    SLT => ...,
    SLTU => ...,
    ...
  }
}
```

```
function Pure_ADD(x, y) -> bits(32)
= {
  let rs1 = <arbitrary index>;
  let rs2 = <arbitrary index>;
  let rd = <arbitrary index>;
  X(rs1) = x;
  X(rs2) = y;
  execute(RTYPE(rs2, rs1, rd, ADD));
  return X(rd);
}
```

How to test / formally verify? (variation 1)



How to test / formally verify? (variation 2)



How to automatically generate VeIR spec?



(And then do all Lean proofs against the simplified .mlir spec)

What needs to be done

- Improve MLIR generation
- Fill gaps in VeIR's dialect support
- Define wrapper templates
- Fill optimization gaps in MLIR / VeIR
- Modify all code generation proofs to use MLIR spec

Start with RISC-V (small/simple)

Extend support to handle x86 and Arm



Final thoughts

- Trustworthy ISA specifications are available – we should use them!
- MLIR conversion useful for other things ...
 - Will benefit from “Functional Languages in MLIR” project
- ISA spec useful for other things <https://alastairreid.github.io/uses-for-isa-specs/>
- Modern ISA specs contain other useful stuff...
 - Machine readable instruction encoding databases
 - Control & Status Register databases
 - Spec of ‘supervisor’ mode (i.e., virtual memory, privilege checks, ...)
 - ...



Questions?

Dialects used in MLIR generation

.isa Language feature	Dialect
Std::Integer	I128 (should be bigint)
Std::Boolean	arith (i1)
Std::Bits(n)	i128 (should be tuple<i16?,bigint>)
Std::RAM	I64 (placeholder for giant array)
Tuples	tuple<>, CIRCT handshake
Records (aka structs)	tuple<>
Arrays	Memref (not ideal!)
Local variables	SSA values
Global variables	Memref (severe type restrictions)

.isa Language feature	Dialect
If/for/while/...	cf.{br,cond_br}
Case stmt	If-elsif-elsif (cf.cond_br)
Exceptions raise/try-catch	cf + tagged unions (but tagged union approximated using tuple 😞)
Functions	func
Dependent types	cf.assert, cf.assume
Debug information	Line numbers
UNSPECIFIED	<not supported yet>

Note: most of these choices are wrong/suboptimal. But they allow me to focus on generation issues without being distracted by defining my own dialect.

MLIR generation

areid-mlir branch of <https://github.com/alastairreid/isa-tools/>

Use / Test generator

Build the compiler, convert a small risc-v spec to MLIR and use mlir-opt to check that the generated mlir is legal

```
# install prerequisites: see [README](https://github.com/alastairreid/isa-tools/blob/master/README.md) on the isa-tools repo (Ubuntu section copied below)
git clone https://github.com/alastairreid/isa-tools.git -b areid-mlir
make
./test.sh
```

riscv/riscv.isa == RISC-V spec

A subset intended for enabling experimentation with things like generating MLIR – the idea being that it is just complicated enough to be interesting, small enough to be manageable and semi-relevant to most people.

If I was creating a full RISC-V spec, I would write it a bit differently – making more use of instruction encoding and register databases.

test2.isa == Test for additional .isa features

Enumerated types, exceptions, tuples/records/structs, ...

std.mlir == Standard library support.

Currently very incomplete

t1.mlir == Generated mlir code

```
function Instr::ADD
    (rs2 : Bits(5), rs1 : Bits(5), rd : Bits(5))
begin
    let s1 := Unsigned(rs1);
    let s2 := Unsigned(rs2);
    let d := Unsigned(rd);
    let operand1 := X[s1];
    let operand2 := X[s2];
    let result := operand1 + operand2;
    X[d] := result;
end
```

iii --exec=':to_mli' **--typecheck'**
mli-opt --inline --canonicalize

```
func.func @Instr$ADD(%arg0: i128, %arg1: i128, %arg2: i128) {
    %c32_i64 = arith.constant 32 : i64
    %c31_i64 = arith.constant 31 : i64
    %c0_i64 = arith.constant 0 : i64
    %c5_i64 = arith.constant 5 : i64
    %0 = call @Std$Bits$MyLength(%arg0) : (i128) -> i64
    %1 = arith.cmpi eq, %0, %c5_i64 : i64
    cf.assert %1, "type assumption"
    %2 = call @Std$Bits$MyLength(%arg1) : (i128) -> i64
    %3 = arith.cmpi eq, %2, %c5_i64 : i64
    cf.assert %3, "type assumption"
    %4 = call @Std$Bits$MyLength(%arg2) : (i128) -> i64
    %5 = arith.cmpi eq, %4, %c5_i64 : i64
    cf.assert %5, "type assumption"
    %6 = call @Std$Bits$MyLength(%arg1) : (i128) -> i64
    %7 = arith.cmpi eq, %6, %c5_i64 : i64
    cf.assert %7, "type assertion"
    %8 = arith.trunci %arg1 : i128 to i63
    %9 = arith.extui %8 : i63 to i64
    %10 = arith.cmpi sle, %9, %c31_i64 : i64
    cf.assert %10, "type assumption"
    %11 = call @Std$Bits$MyLength(%arg0) : (i128) -> i64
    %12 = arith.cmpi eq, %11, %c5_i64 : i64
    cf.assert %12, "type assertion"
    %13 = arith.trunci %arg0 : i128 to i63
    %14 = arith.extui %13 : i63 to i64
    %15 = arith.cmpi sle, %14, %c31_i64 : i64
    cf.assert %15, "type assumption"
    %16 = call @Std$Bits$MyLength(%arg2) : (i128) -> i64
    %17 = arith.cmpi eq, %16, %c5_i64 : i64
    cf.assert %17, "type assertion"
    %18 = arith.trunci %arg2 : i128 to i63
    %19 = arith.extui %18 : i63 to i64
    %20 = arith.cmpi sle, %19, %c31_i64 : i64
    cf.assert %20, "type assumption"
    %21 = arith.cmpi sle, %9, %c31_i64 : i64
    cf.assert %21, "type assertion"
    %22 = arith.cmpi sle, %9, %c31_i64 : i64
    cf.assert %22, "type assumption"
    %23 = arith.cmpi eq, %9, %c0_i64 : i64
    cf.cond_br %23, ^bb1, ^bb2
    ^bb1: // pred: ^bb0
    %24 = call @Std$Bits$Zero(%c32_i64) : (i64) -> i128
    %25 = call @Std$Bits$MyLength(%24) : (i128) -> i64
    %26 = arith.cmpi eq, %25, %c32_i64 : i64
    cf.assert %26, "type assumption"
    cf.br ^bb3(%24 : i128)
    ^bb2: // pred: ^bb0
    %27 = index.casts %9 : i64 to index
    %28 = memref.get_global @_X : memref<32xi128>
    %29 = memref.load %28[%27] : memref<32xi128>
    cf.br ^bb3(%29 : i128)
    ^bb3(%30: i128): // 2 preds: ^bb1, ^bb2
    %31 = call @Std$Bits$MyLength(%30) : (i128) -> i64
    %32 = memref.get_global @XLEN : memref<i64>
    %33 = memref.load %32[] : memref<i64>
    %34 = arith.cmpi eq, %31, %33 : i64
    cf.assert %34, "type assertion"
    %35 = call @Std$Bits$MyLength(%30) : (i128) -> i64
    %36 = memref.get_global @XLEN : memref<i64>
    %37 = memref.load %36[] : memref<i64>
    %38 = arith.cmpi eq, %35, %37 : i64
    cf.assert %38, "type assumption"
    %39 = arith.cmpi sle, %14, %c31_i64 : i64
    cf.assert %39, "type assertion"
    %40 = arith.cmpi sle, %14, %c31_i64 : i64
    cf.assert %40, "type assumption"
    %41 = arith.cmpi eq, %14, %c0_i64 : i64
    cf.cond_br %41, ^bb4, ^bb5
}
```

```
^bb4: // pred: ^bb3
%42 = call @Std$Bits$Zero(%c32_i64) : (i64) -> i128
%43 = call @Std$Bits$MyLength(%42) : (i128) -> i64
%44 = arith.cmpi eq, %43, %c32_i64 : i64
cf.assert %44, "type assumption"
cf.br ^bb6(%42 : i128)
^bb5: // pred: ^bb3
%45 = index.casts %14 : i64 to index
%46 = memref.get_global @_X : memref<32xi128>
%47 = memref.load %46[%45] : memref<32xi128>
cf.br ^bb6(%47 : i128)
^bb6(%48: i128): // 2 preds: ^bb4, ^bb5
%49 = call @Std$Bits$MyLength(%48) : (i128) -> i64
%50 = memref.get_global @XLEN : memref<i64>
%51 = memref.load %50[] : memref<i64>
%52 = arith.cmpi eq, %49, %51 : i64
cf.assert %52, "type assertion"
%53 = call @Std$Bits$MyLength(%48) : (i128) -> i64
%54 = memref.get_global @XLEN : memref<i64>
%55 = memref.load %54[] : memref<i64>
%56 = arith.cmpi eq, %53, %55 : i64
cf.assert %56, "type assumption"
%57 = call @Std$Bits$MyLength(%30) : (i128) -> i64
%58 = arith.cmpi eq, %57, %c32_i64 : i64
cf.assert %58, "type assertion"
%59 = call @Std$Bits$MyLength(%48) : (i128) -> i64
%60 = arith.cmpi eq, %59, %c32_i64 : i64
cf.assert %60, "type assertion"
%61 = arith.addi %30, %48 : i128
%62 = call @Std$Bits$MyLength(%61) : (i128) -> i64
%63 = arith.cmpi eq, %62, %c32_i64 : i64
cf.assert %63, "type assumption"
%64 = arith.cmpi sle, %19, %c31_i64 : i64
cf.assert %64, "type assertion"
%65 = call @Std$Bits$MyLength(%61) : (i128) -> i64
%66 = memref.get_global @XLEN : memref<i64>
%67 = memref.load %66[] : memref<i64>
%68 = arith.cmpi eq, %65, %67 : i64
cf.assert %68, "type assertion"
%69 = arith.cmpi sle, %19, %c31_i64 : i64
cf.assert %69, "type assumption"
%70 = call @Std$Bits$MyLength(%61) : (i128) -> i64
%71 = memref.get_global @XLEN : memref<i64>
%72 = memref.load %71[] : memref<i64>
%73 = arith.cmpi eq, %70, %72 : i64
cf.assert %73, "type assumption"
%74 = arith.cmpi ne, %19, %c0_i64 : i64
cf.cond_br %74, ^bb7, ^bb8
^bb7: // pred: ^bb6
%75 = index.casts %19 : i64 to index
%76 = memref.get_global @_X : memref<32xi128>
memref.store %61, %76[%75] : memref<32xi128>
cf.br ^bb8
^bb8: // 2 preds: ^bb6, ^bb7
return
}
```