MLIR <-> ISA Specifications (WIP)

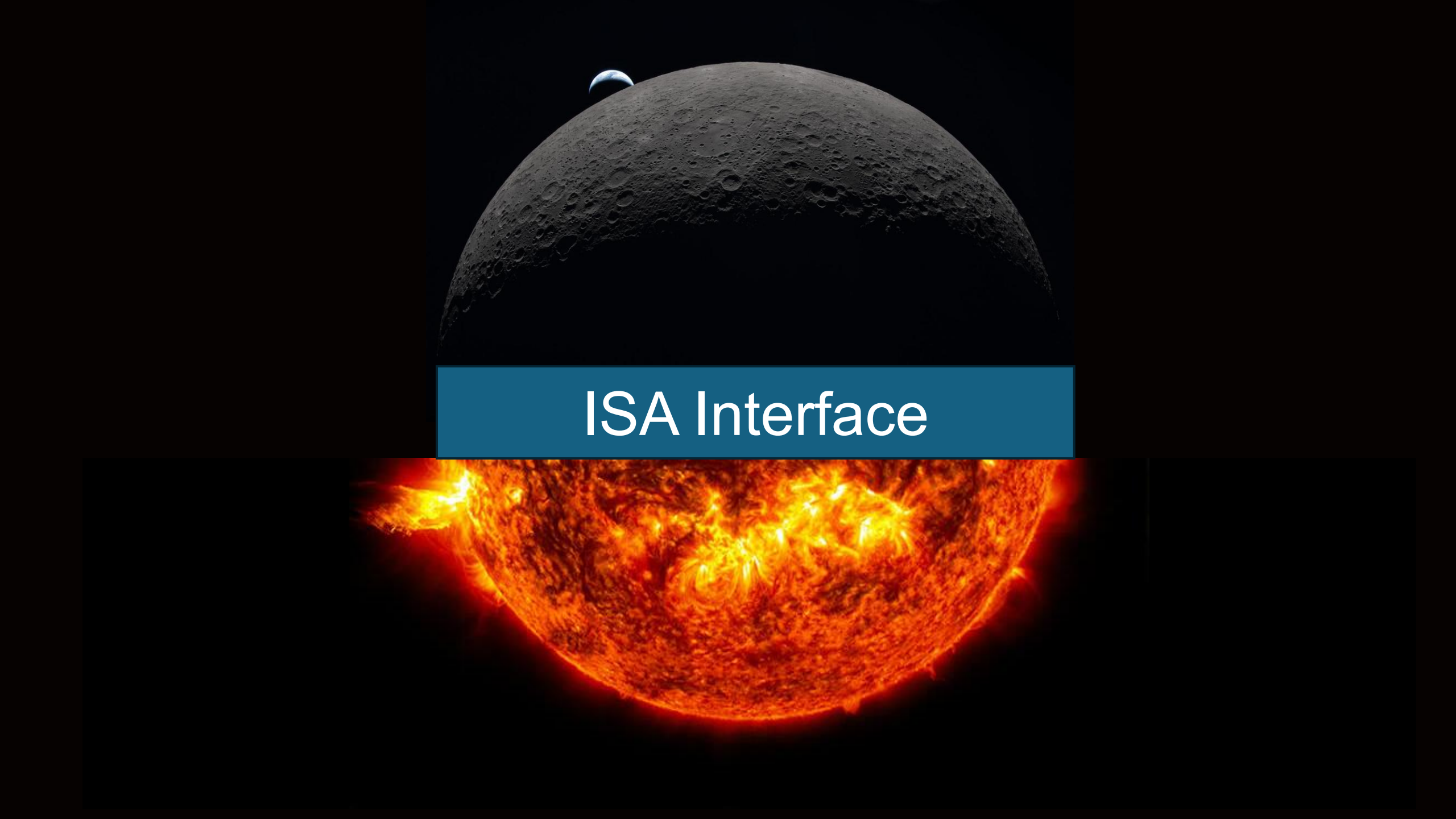
Alastair Reid

Intel

(Don’t blame Sasha Lopoukhine or Mathieu Fehr)

14 August 2026,

MLIR 2026 Summer School, A Coruña, Spain



ISA Interface

Readable,
trustworthy
documentation

Compilers
- Test
- Verify
- Generate

OSes
- …

CPUs / GPUs
- …

State of the art

	Intel x86	Intel GPU	RISC-V	Arm AArch32	Arm AArch64	Arm M-class
Executable & Tested	Yes	Yes	Yes	Yes	Yes	Yes
In official (public) documentation	Soon	No	Soon?	Yes	Yes	Yes
Machine-readable spec available	Sooner	No	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

.isa example (Another 50kloc omitted)

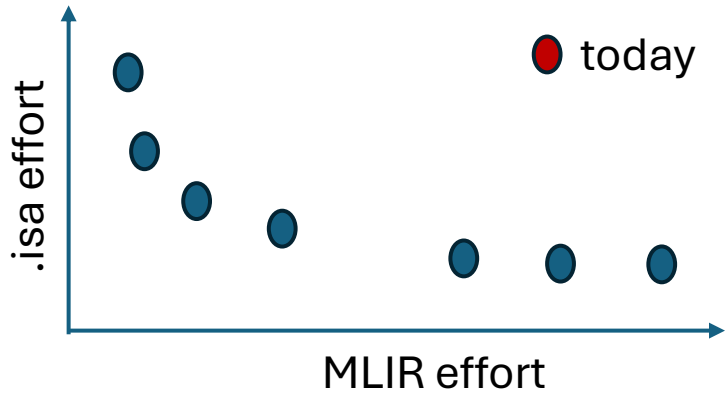
```
function Count_Leading_Zero(size : {0..}, x : Bits(size)) -> {0 .. size}
begin
   var count : {0..size} = 0;
   for i := size-1 downto 0 do
      if x[i] == 0b1 then
         return count;
      endif;
      count := count + 1;
   endfor;
   assert count == size;
   return count;
end
```

Dependent Types
(i.e., type depends on a value)
Value constraints
Early function return
Immutable variables
Mutable variables
Pattern matching case statement

Design criteria for generating MLIR

- Easy to generate from .isa? (#loc)
- Easy to use in MLIR?
  - To generate LLVM dialect (#loc)
  - To generate smt dialect (#loc)
  - …

  (Can we use standard MLIR dialects, passes, etc.?)
- Preserve as much high-level information as possible
  - Test: “convert; unconvert; convert == convert”? (Projection/embedding pair)
- Can any new dialects be used for .asl, .sail, …
  - Will they still work in 5 years time?



Recommendation of
Sasha Lopoukhine

```
function CLZ(size : {0..}, x :
Bits(size))
                        -> {0 .. size}
begin
   var count : {0..size} := 0;
   for i := size-1 downto 0 do
      if x[i] == 0b1 then
         return count;
      endif;
      count := count + 1;
   endfor;
   assert count == size;
   return count;
end
```

```
function CLZ(size : {0..}, x : Bits(size))
                        -> {0 .. size}
begin
   var count : {0..size} := 0;
   for i := size-1 downto 0 do
      if x[i] == 0b1 then
         return count;
      endif;
      count := count + 1;
   endfor;
   assert count == size;
   return count;
end
```

```
func.func @CLZ(%size: ??, %x : ??) -> ?? {
  %zero = ??.constant 0 : ??.integer
  %count = ??.alloca %zero : ??.ref;
  ??.for(...) {
     %c = ??.load %count : ??.integer
     %0 = ??.slice %x, %i : ??.integer
     %1 = ??.cmpeq %0, %one : i1
     ??.if %1 {
        ??.return %c : ??.integer
     }
     %c2 = ??.add %c, %one : ??.integer
     ??.store %count, %c2
   }
   %2 = ??.cmpeq %count, %size : i1
   cf.assert %2
   ??.return %count : ??.integer
}
```

```
function CLZ(size : {0..}, x : Bits(size))
                        -> {0 .. size}
begin
   var count : {0..size} := 0;
   for i := size-1 downto 0 do
      if x[i] == 0b1 then
         return count;
      endif;
      count := count + 1;
   endfor;
   assert count == size;
   return count;
end
```

Custom Dialects:

- bigint

```
func.func @CLZ(%size: bigint.int, %x : ??) -> bigint {
  %zero = bigint.constant 0 : bigint.int
  %count = bigint.alloca %zero : bigint.ref;
  ??.for(...) {
     %c = bigint.load %count : bigint.int
     %0 = ??.slice %x, %i : bigint.int
     %1 = bigint.cmpeq %0, %one : i1
     ??.if %1 {
        ??.return %c : bigint.int
     }
     %c2 = bigint.add %c, %one : bigint.int
     bigint.store %count, %c2
   }
   %2 = bigint.cmpeq %count, %size : i1
   cf.assert %2
   ??.return %count : bigint.int
}
```

```
function CLZ(size : {0..}, x : Bits(size))
                        -> {0 .. size}
begin
   var count : {0..size} := 0;
   for i := size-1 downto 0 do
      if x[i] == 0b1 then
         return count;
      endif;
      count := count + 1;
   endfor;
   assert count == size;
   return count;
end
```

Custom Dialects:

- bigint – unbounded ints
- me_scf – SCF with “Multiple Exits”

```
func.func @CLZ(%size: bigint.int, %x : ??) -> bigint {
  %zero = bigint.constant 0 : bigint.int
  %count = bigint.alloca %zero : bigint.ref;
  me_scf.for(...) {
     %c = bigint.load %count : bigint.int
     %0 = ??.slice %x, %i : bigint.int
     %1 = bigint.cmpeq %0, %one : i1
     me_scf.if %1 {
        me_scf.return %c : bigint.int
     }
     %c2 = bigint.add %c, %one : bigint.int
     bigint.store %count, %c2
   }
   %2 = bigint.cmpeq %count, %size : i1
   cf.assert %2
   me_scf.return %count : bigint.int
}
```

```
function CLZ(size : {0..}, x : Bits(size))
                    -> {0 .. size}
begin
  var count : {0..size} := 0;
  for i := size-1 downto 0 do
    if x[i] == 0b1 then
      return count;
    endif;
    count := count + 1;
  endfor;
  assert count == size;
  return count;
end
```

Custom Dialects:

- bigint – unbounded ints
- me_scf – SCF with “Multiple Exits”

```
func.func @CLZ(%size: bigint.int, %x : ??) -> bigint {
  %zero = bigint.constant 0 : bigint.int
  %count = bigint.alloca %zero : bigint.ref;
  me_scf.for(...) {
    %c = bigint.load %count : bigint.int
    %0 = ??.slice %x, %i : bigint.int
    %1 = bigint.cmpeq %0, %one : i1
    me_scf.if %1 {
      me_scf.return %c : bigint.int
    }
    %c2 = bigint.add %c, %one : bigint.int
    bigint.store %count, %c2
  }
  %2 = bigint.cmpeq %count, %size : i1
  cf.assert %2
  me_scf.return %count : bigint.int
}
```

```
function CLZ(size : {0..}, x : Bits(size))
                -> {0 .. size}
begin
  var count : {0..size} := 0;
  for i := size-1 downto 0 do
    if x[i] == 0b1 then
      return count;
    endif;
    count := count + 1;
  endfor;
  assert count == size;
  return count;
end
```

Custom Dialects:

- bigint – unbounded ints
- me_scf – SCF with “Multiple Exits”

```
func.func @CLZ(%size: bigint.int, %x : ??) -> bigint {
  %nonneg = bigint.cmpge %size, %zero
  cf.assert %nonneg
 %zero = bigint.constant 0 : bigint.int
 %count = bigint.alloca %zero : bigint.ref;
 me_scf.for(...) {
    %c = bigint.load %count : bigint.int
    %0 = ??.slice %x, %i : bigint.int
    %1 = bigint.cmpeq %0, %one : i1
    me_scf.if %1 {
        %ok2 = bigint.cmple %result, %size
        cf.assert %ok2
      me_scf.return %c : bigint.int
    }
    %c2 = bigint.add %c, %one : bigint.int
    bigint.store %count, %c2
  }
  %2 = bigint.cmpeq %count, %size : i1
  cf.assert %2
   %ok2 = bigint.cmple %result, %size
   cf.assert %ok2
  me_scf.return %count : bigint.int
}
```

```
function CLZ(size : {0..}, x : Bits(size))
                        -> {0 .. size}
begin
   var count : {0..size} := 0;
   for i := size-1 downto 0 do
      if x[i] == 0b1 then
         return count;
      endif;
      count := count + 1;
   endfor;
   assert count == size;
   return count;
end
```

Custom Dialects:

- bigint – unbounded ints
- me_scf – SCF with “Multiple Exits”

```
func.func @CLZ(%size: bigint.int, %x : ??) -> bigint {
  %zero = bigint.constant 0 : bigint.int
  %count = bigint.alloca %zero : bigint.ref;
  me_scf.for(...) {
     %c = bigint.load %count : bigint.int
     %0 = ??.slice %x, %i : bigint.int
     %1 = bigint.cmpeq %0, %one : i1
     me_scf.if %1 {
        me_scf.return %c : bigint.int
     }
     %c2 = bigint.add %c, %one : bigint.int
     bigint.store %count, %c2
  }
  %2 = bigint.cmpeq %count, %size : i1
  cf.assert %2
  me_scf.return %count : bigint.int
}
```

Credit: NASA

Option #1 (of 2)
Dynamic typing

Custom Dialects:

- bigint – unbounded ints
- me_scf – SCF with “Multiple Exits”
- bitvector – bitvectors (== bigint?)

```
func.func @CLZ(%size: bigint.int, %x : bitvector) -> bigint {
  %length = bitvector.length %x : bigint.int
  %chk1 = bigint.cmpeq %size, bigint.length(%x) : i1
  cf.assert %chk1
  %zero = bigint.constant 0 : bigint.int
  %count = bigint.alloca %zero : bigint.ref;
  me_scf.for(...) {
    %c = bigint.load %count : bigint.int
    %0 = bitvector.slice %x, %i : bigint.int
    %1 = bigint.cmpeq %0, %one : i1
    me_scf.if %1 {
      me_scf.return %c : bigint.int
    }
    %c2 = bigint.add %c, %one : bigint.int
    bigint.store %count, %c2
  }
  %2 = bigint.cmpeq %count, %size : i1
  cf.assert %2
  me_scf.return %count : bigint.int
}
```

Option #2 (of 2)
Types are values too!

Custom Dialects:

- bigint – unbounded ints
- me_scf – SCF with “Multiple Exits”
- bitvector – bitvectors
- deptype – dependent type support

```
func.func @CLZ(%type_of_x : type, %x : deptype) -> bigint
{
  %zero = bigint.constant 0 : bigint.int
  %count = bigint.alloca %zero : bigint.ref;
  me_scf.for(...) {
      %c = bigint.load %count : bigint.int
      %0 = bitvector.slice %x, %i : bigint.int
      %1 = bigint.cmpeq %0, %one : i1
      me_scf.if %1 {
          me_scf.return %c : bigint.int
      }
      %c2 = bigint.add %c, %one : bigint.int
      bigint.store %count, %c2
    }
    %2 = bigint.cmpeq %count, %size : i1
    cf.assert %2
    me_scf.return %count : bigint.int
}
```

Summary

- MLIR + ISA = 

- Design criteria (with conflicts)
- We need a flexible solution
- Need some custom dialects
- Not sure of path forward: dynamic typing vs. dependent typing?
- WIP – but my progress has been quite slow…