Hardware Compilation with MLIR and CIRCT

Hanchen Ye, Advisor: Jin Yang

hanchenye@gmail.com

July 28, 2022

About me

Hanchen Ye is an PhD candidate in ECE at UIUC advised by professor Deming Chen. He obtained his Bachelor and Master degree at Fudan University in 2017 and 2019, respectively. His researches high-level synthesis (HLS), domain-specific compilers, and hardware acceleration. He has published multiple conference papers on DAC, ICCAD, HPCA, etc. He has been leading or contributing to open-source projects, including ScaleHLS, MLIR, CIRCT, MLIR-AIE, etc.

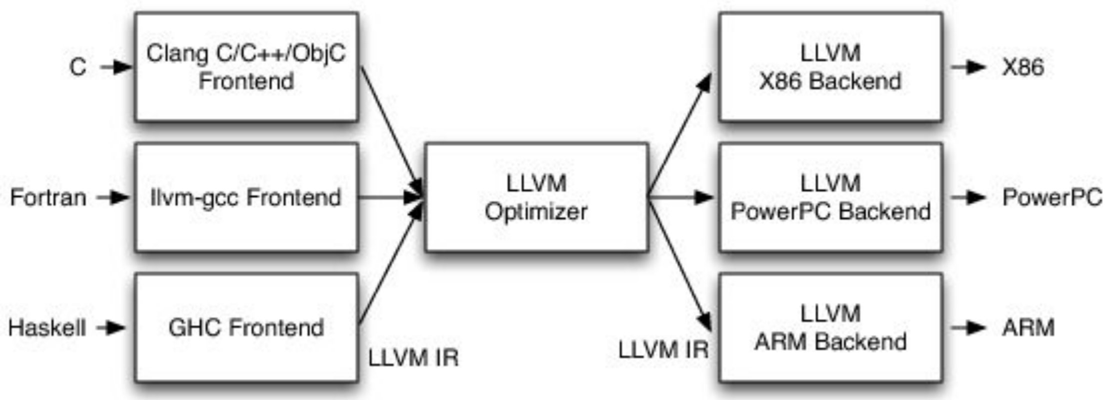
Outline

- **Background:** From LLVM to MLIR
- **MLIR:** Multi-Level Intermediate Representation
- **CIRCT:** Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

Outline

- **Background:** From LLVM to MLIR
- **MLIR:** Multi-Level Intermediate Representation
- **CIRCT:** Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

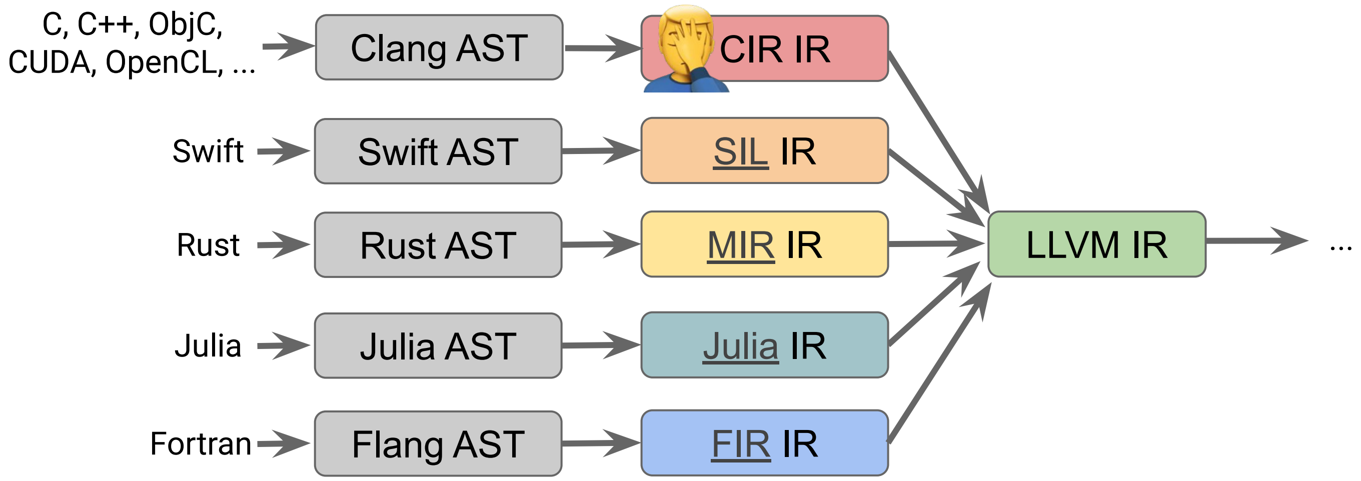
From LLVM to MLIR



- LLVM uses the same intermediate representation (IR) to represent ALL programs.
- All program optimizations are based on the LLVM IR.
- LLVM dispatches the front-ends, optimizations, and back-ends. O(m*n) -> O(1)

Source: The architecture of open-source applications, C. Lattner.

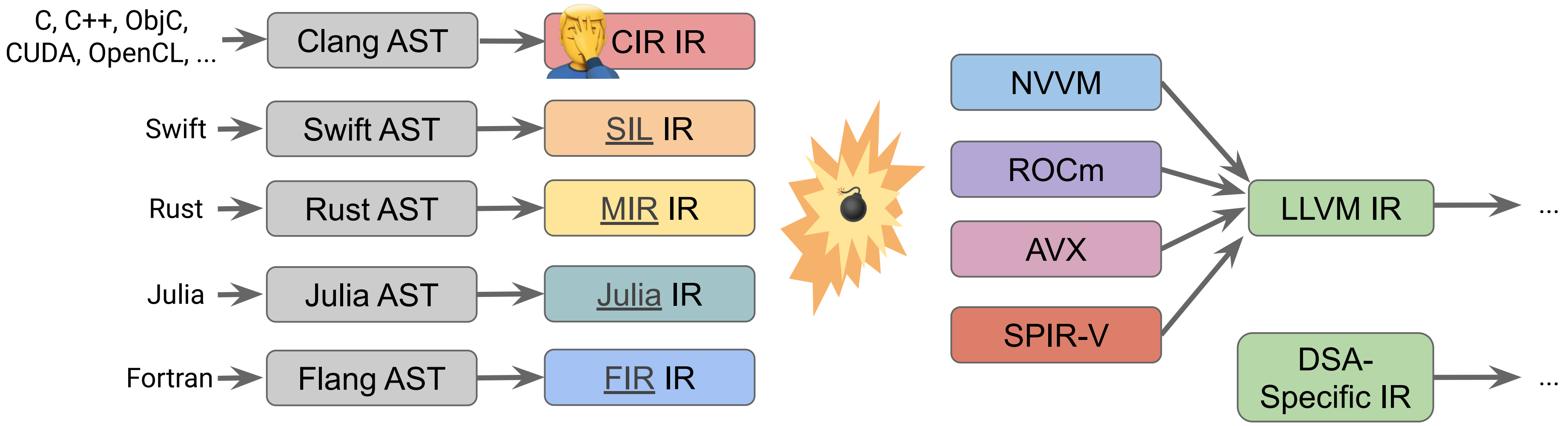
From LLVM to MLIR (Cont'd)



- More and more programming languages demand customized IR for optimization.
- The IR for different languages have different abstraction level.
- Language-specific IR can be lowered to LLVM for back-end code generation.

Source: MLIR: Multi-Level Intermediate Representation Compiler Infrastructure, C. Lattner.

From LLVM to MLIR (Cont'd)



- Different back-ends demand customized IR for optimization
- DSAs even cannot use LLVM for generating back-end codes and demand their own IR for code generation

Severe Fragmentation: IRs have different implementations and "frameworks"

MLIR: Compiler Infrastructure for the End of Moore’s Law

- **M**ulti-**L**evel **I**ntermediate **R**epresentation
- State of the art compiler technology
- Built on top of LLVM’s open and library-based philosophy
- **Modular and extensible**
- Originally created within Google for compiling TensorFlow
- **Sufficiently general** to compile lots of domains

https://mlir.llvm.org

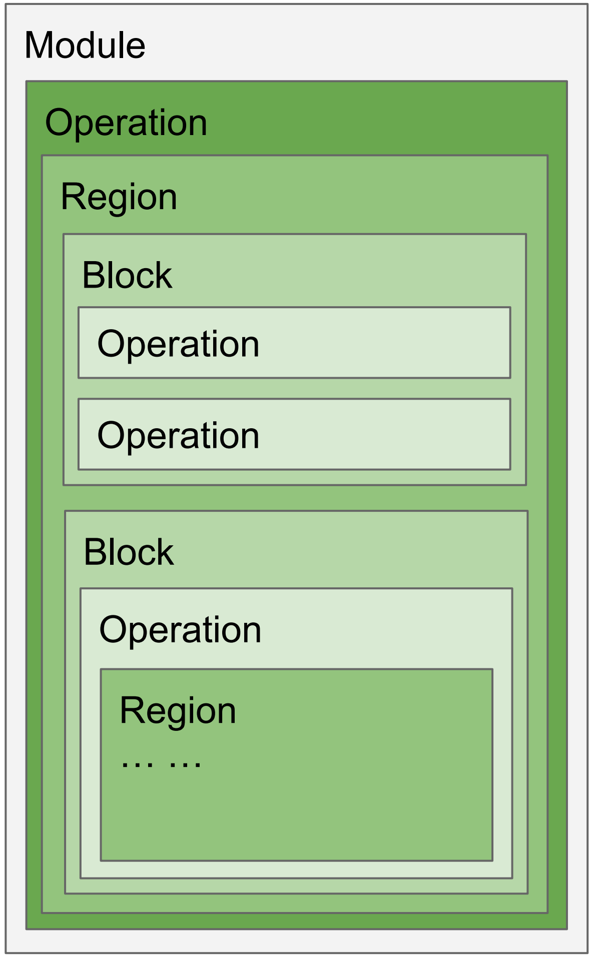
Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Syntax of MLIR

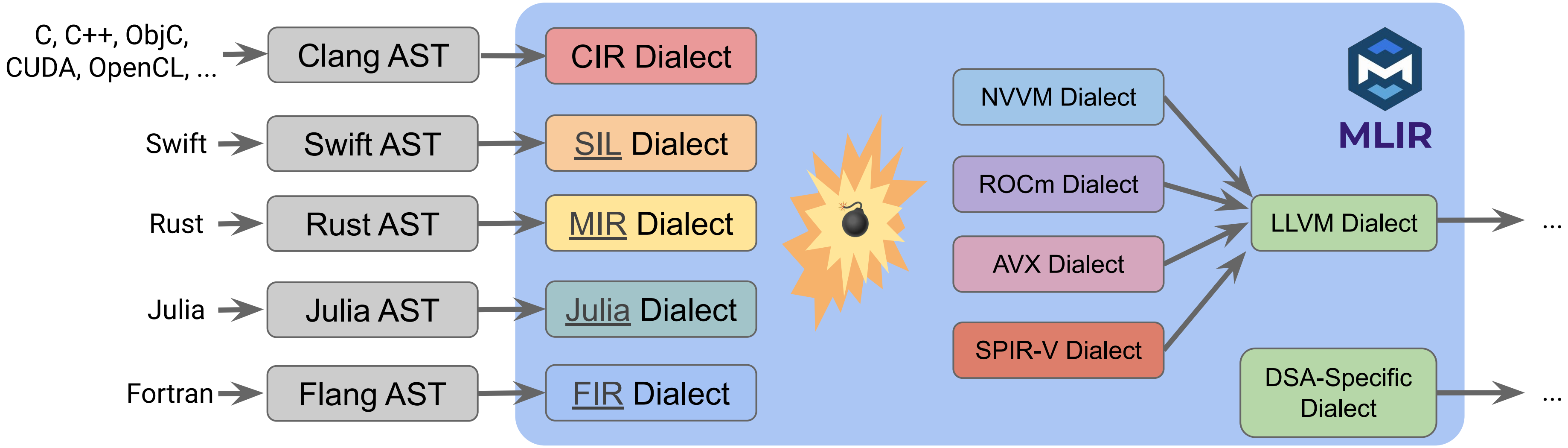
- SSA-based IR design, explicit typing system
- Module/Operation/Region/Block/Operation hierarchy
- Operation can contain multiple Regions

```
func.func @testFunction(%arg0: i32) -> i32 {
  %a = func.call @thingToCall(%arg0) : (i32) -> i32
  cf.br ^bb1
^bb1:
  %c = affine.for %i = 0 to 10 iter_args(%b = %a) -> i32 {
    %i_i32 = arith.index_cast %i : index to i32
    %b_new = arith.addi %i_i32, %b : i32
    affine.yield %b_new : i32
  }
  func.return %c : i32
}
```

Dialect A C++ namespace that contains customized operations, types, and attributes. Implement the “correct” abstraction for your domain.



MLIR: “Meta IR” and Compiler Infrastructure



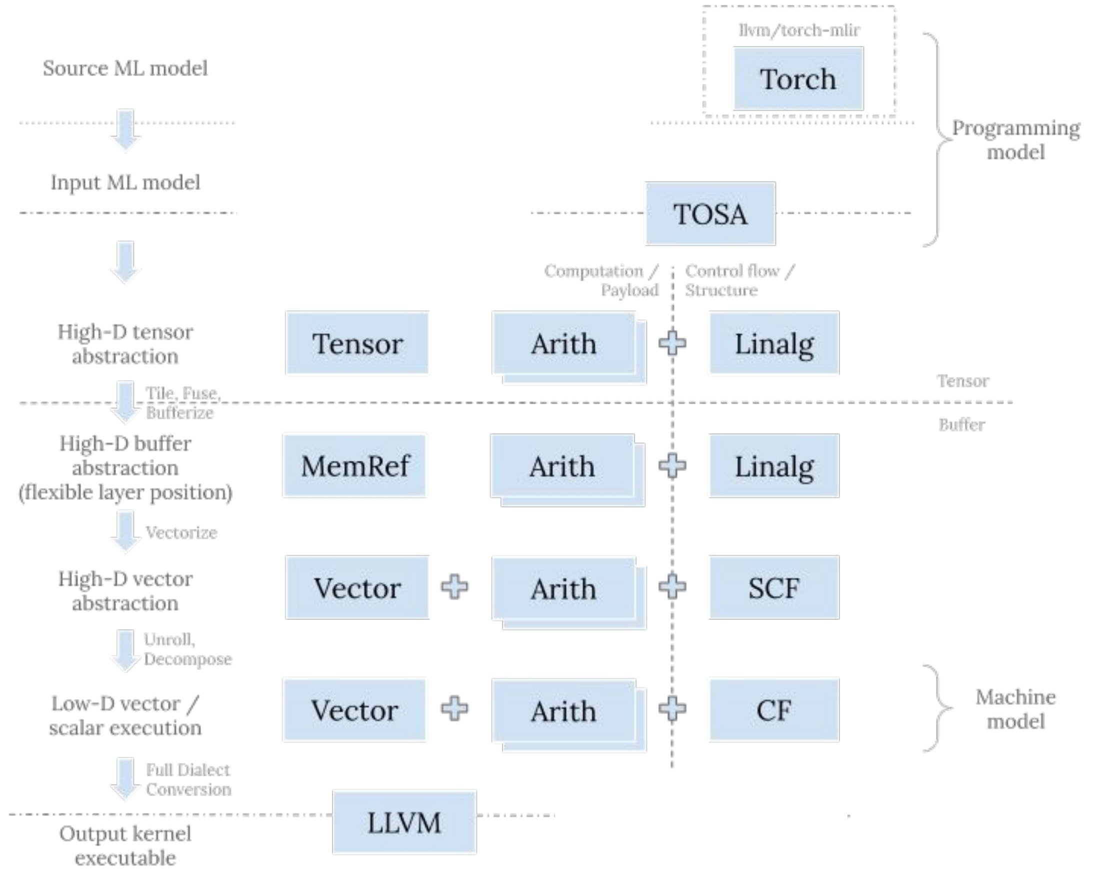
MLIR is a **“Meta IR”** and **compiler infrastructure** for:

- Design and implement **dialect**
- Optimization and transform inside of a **dialect**
- Conversion between different **dialects**
- Code generation of **dialect**

Outline

- **Background:** From LLVM to MLIR
- **MLIR:** Multi-Level Intermediate Representation
- **CIRCT:** Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

MLIR Dialects Walk-through: Progressive Lowering



Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, L. Zhang.

ML Model Compilation: PyTorch to Torch Dialect

```
class Linear(nn.Module):
    def __init__(self):
        super(Linear, self).__init__()
        self.linear = nn.Linear(16, 10)

    def forward(self, x):
        return self.linear(x)


linear = Linear()
mlir_module = torch_mlir.compile(linear, torch.ones(
    1, 16), output_type=torch_mlir.OutputType.TORCH)
```

PyTorch Model

Torch-MLIR Front-end

```
func.func @forward(%arg0: !torch.vtensor<[1,16],f32>) -> !torch.vtensor<[1,10],f32> {
  %0 = torch.vtensor.literal(dense<"0xA270..."> : tensor<10xf32>) : !torch.vtensor<[10],f32>
  %1 = torch.vtensor.literal(dense<"0x5CE5..."> : tensor<10x16xf32>) : !torch.vtensor<[10,16],f32>
  %2 = torch.aten.linear %arg0, %1, %0 : !torch.vtensor<[1,16],f32>, !torch.vtensor<[10,16],f32>,
!torch.vtensor<[10],f32> -> !torch.vtensor<[1,10],f32>
  return %2 : !torch.vtensor<[1,10],f32>
}
```

Torch Dialect

Torch Dialect

- Front-end dialect designed for interfacing PyTorch and MLIR.
- This dialect maintains a fairly isomorphic representation with TorchScript.
- Operates on tensor objects with static ranks inferred where possible and propagated throughout the program.

Source: Torch-MLIR https://github.com/llvm/torch-mlir

ML Model Compilation: Torch to TOSA

Torch to TOSA Lowering

```
func.func @forward(%arg0: tensor<1x16xf32>) -> tensor<1x10xf32> {
  %0 = "tosa.const"() {value = dense<"0xC44B..."> : tensor<1x16x10xf32>} : () -> tensor<1x16x10xf32>
  %1 = "tosa.const"() {value = dense<"0xA270..."> : tensor<1x10xf32>} : () -> tensor<1x10xf32>
  %2 = "tosa.reshape"(%arg0) {new_shape = [1, 1, 16]} : (tensor<1x16xf32>) -> tensor<1x1x16xf32>
  %3 = "tosa.matmul"(%2, %0) : (tensor<1x1x16xf32>, tensor<1x16x10xf32>) -> tensor<1x1x10xf32>
  %4 = "tosa.reshape"(%3) {new_shape = [1, 10]} : (tensor<1x1x10xf32>) -> tensor<1x10xf32>
  %5 = "tosa.add"(%4, %1) : (tensor<1x10xf32>, tensor<1x10xf32>) -> tensor<1x10xf32>
  return %5 : tensor<1x10xf32>
}
```

Torch Dialect

TOSA (Tensor Operators Set Architecture) Dialect

- A front-end and back-end agnostic dialect representing a minimal and stable set of tensor-level operations commonly employed by Machine Learning frameworks.
- Detailed functional and numerical description enabling precise code construction for a diverse range of targets – SIMD CPUs, GPUs and custom domain-specific accelerators.

Source: Torch-MLIR https://github.com/llvm/torch-mlir

ML Model Compilation: TOSA to Linalg on Tensors

⇩ *TOSA to Linalg Lowering*

```
#map0 = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
func.func @forward(%arg0: tensor<1x16xf32>) -> tensor<1x10xf32> {
  %cst = arith.constant dense<"0xA270..."> : tensor<1x10xf32>
  %cst_0 = arith.constant dense<"0xC44B..."> : tensor<16x10xf32>
  %0 = linalg.generic {indexing_maps = [#map0, #map1, #map2], iterator_types = ["parallel", "parallel", "reduction"]}
ins(%arg0, %cst_0 : tensor<1x16xf32>, tensor<16x10xf32>) outs(%cst : tensor<1x10xf32>) {
  ^bb0(%arg1: f32, %arg2: f32, %arg3: f32):
    %1 = arith.mulf %arg1, %arg2 : f32
    %2 = arith.addf %arg3, %1 : f32
    linalg.yield %2 : f32
  } -> tensor<1x10xf32>
  return %0 : tensor<1x10xf32>
}
```

Tensor + Arith + Linalg Dialects

Linalg (Linear Algebra) Dialect

- High-level and structured representation of linear algebra operators
- Designed for driving transformations including buffer allocation, parametric tiling, vectorization, etc.

ML Model Compilation: Bufferization

Func, Arith, and Linalg Bufferization

```
#map0 = affine_map<(d0, d1, d2) -> (d0, d2)>
#map1 = affine_map<(d0, d1, d2) -> (d2, d1)>
#map2 = affine_map<(d0, d1, d2) -> (d0, d1)>
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %2 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  linalg.generic {indexing_maps = [#map0, #map1, #map2], iterator_types = ["parallel", "parallel", "reduction"]}
ins(%arg0, %2 : memref<1x16xf32>, memref<16x10xf32>) outs(%arg1 : memref<1x10xf32>) {
  ^bb0(%arg2: f32, %arg3: f32, %arg4: f32):
    %3 = arith.mulf %arg2, %arg3 : f32
    %4 = arith.addf %arg4, %3 : f32
    linalg.yield %4 : f32
  }
  return
}
```

Memref + Arith + Linalg Dialects

ML Model Compilation: Linalg to Affine

Linalg to Affine Lowering

```
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %1 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  affine.for %arg2 = 0 to 10 {
    affine.for %arg3 = 0 to 16 {
      %2 = affine.load %arg0[0, %arg3] : memref<1x16xf32>
      %3 = affine.load %1[%arg3, %arg2] : memref<16x10xf32>
      %4 = affine.load %arg1[0, %arg2] : memref<1x10xf32>
      %5 = arith.mulf %2, %3 : f32
      %6 = arith.addf %4, %5 : f32
      affine.store %6, %arg1[0, %arg2] : memref<1x10xf32>
    }
  }
  return
}
```

Memref + Arith + Affine Dialects

Affine Dialect

Designed for using techniques from polyhedral compilation to make dependence analysis and loop transformations efficient and reliable.

ML Model Compilation: Affine-level Vectorization

Affine Super Vectorization

```
#map = affine_map<(d0, d1) -> (0)>
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %c0 = arith.constant 0 : index
  %cst = arith.constant 0.000000e+00 : f32
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %1 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  affine.for %arg2 = 0 to 10 step 2 {
    affine.for %arg3 = 0 to 16 {
      %2 = vector.transfer_read %arg0[%c0, %arg3], %cst {permutation_map = #map} : memref<1x16xf32>, vector<2xf32>
      %3 = vector.transfer_read %1[%arg3, %arg2], %cst : memref<16x10xf32>, vector<2xf32>
      %4 = vector.transfer_read %arg1[%c0, %arg2], %cst : memref<1x10xf32>, vector<2xf32>
      %5 = arith.mulf %2, %3 : vector<2xf32>
      %6 = arith.addf %4, %5 : vector<2xf32>
      vector.transfer_write %6, %arg1[%c0, %arg2] : vector<2xf32>, memref<1x10xf32>
    }
  }
  return
}
```

Memref + Vector + Arith + Affine Dialects

ML Model Compilation: Affine to SCF

⇩ *Affine to SCF Lowering*

```
#map = affine_map<(d0, d1) -> (0)>
memref.global "private" constant @__constant_16x10xf32 : memref<16x10xf32> = dense<"0xC44B...">
memref.global "private" constant @__constant_1x10xf32 : memref<1x10xf32> = dense<"0xA270...">
func.func @forward(%arg0: memref<1x16xf32>, %arg1: memref<1x10xf32>) {
  %c1 = arith.constant 1 : index    %c16 = arith.constant 16 : index    %c2 = arith.constant 2 : index
  %c10 = arith.constant 10 : index    %c0 = arith.constant 0 : index    %cst = arith.constant 0.000000e+00 : f32
  %0 = memref.get_global @__constant_1x10xf32 : memref<1x10xf32>
  %1 = memref.get_global @__constant_16x10xf32 : memref<16x10xf32>
  memref.copy %0, %arg1 : memref<1x10xf32> to memref<1x10xf32>
  scf.for %arg2 = %c0 to %c10 step %c2 {
    scf.for %arg3 = %c0 to %c16 step %c1 {
      %2 = vector.transfer_read %arg0[%c0, %arg3], %cst {permutation_map = #map} : memref<1x16xf32>, vector<2xf32>
      %3 = vector.transfer_read %1[%arg3, %arg2], %cst : memref<16x10xf32>, vector<2xf32>
      %4 = vector.transfer_read %arg1[%c0, %arg2], %cst : memref<1x10xf32>, vector<2xf32>
      %5 = arith.mulf %2, %3 : vector<2xf32>
      %6 = arith.addf %4, %5 : vector<2xf32>
      vector.transfer_write %6, %arg1[%c0, %arg2] : vector<2xf32>, memref<1x10xf32>
    }
  }
  return
}
```

Memref + Vector + Arith + SCF Dialects

SCF Dialect
Represents structured control flow.

ML Model Compilation: SCF to CF (Control Flow)

⇩ *SCF to CF Lowering*

```
  ... ...
^bb1(%2: index):  // 2 preds: ^bb0, ^bb4
  %3 = arith.cmpi slt, %2, %c10 : index
  cf.cond_br %3, ^bb2(%c0 : index), ^bb5
^bb2(%4: index):  // 2 preds: ^bb1, ^bb3
  %5 = arith.cmpi slt, %4, %c16 : index
  cf.cond_br %5, ^bb3, ^bb4
^bb3:  // pred: ^bb2
  %6 = vector.transfer_read %arg0[%c0, %4], %cst {permutation_map = #map} : memref<1x16xf32>, vector<2xf32>
  %7 = vector.transfer_read %1[%4, %2], %cst : memref<16x10xf32>, vector<2xf32>
  %8 = vector.transfer_read %arg1[%c0, %2], %cst : memref<1x10xf32>, vector<2xf32>
  %9 = arith.mulf %6, %7 : vector<2xf32>
  %10 = arith.addf %8, %9 : vector<2xf32>
  vector.transfer_write %10, %arg1[%c0, %2] : vector<2xf32>, memref<1x10xf32>
  %11 = arith.addi %4, %c1 : index
  cf.br ^bb2(%11 : index)
^bb4:  // pred: ^bb2
  %12 = arith.addi %2, %c2 : index
  cf.br ^bb1(%12 : index)
^bb5:  // pred: ^bb1
  return
}
```

Memref + Vector + Arith + CF Dialects

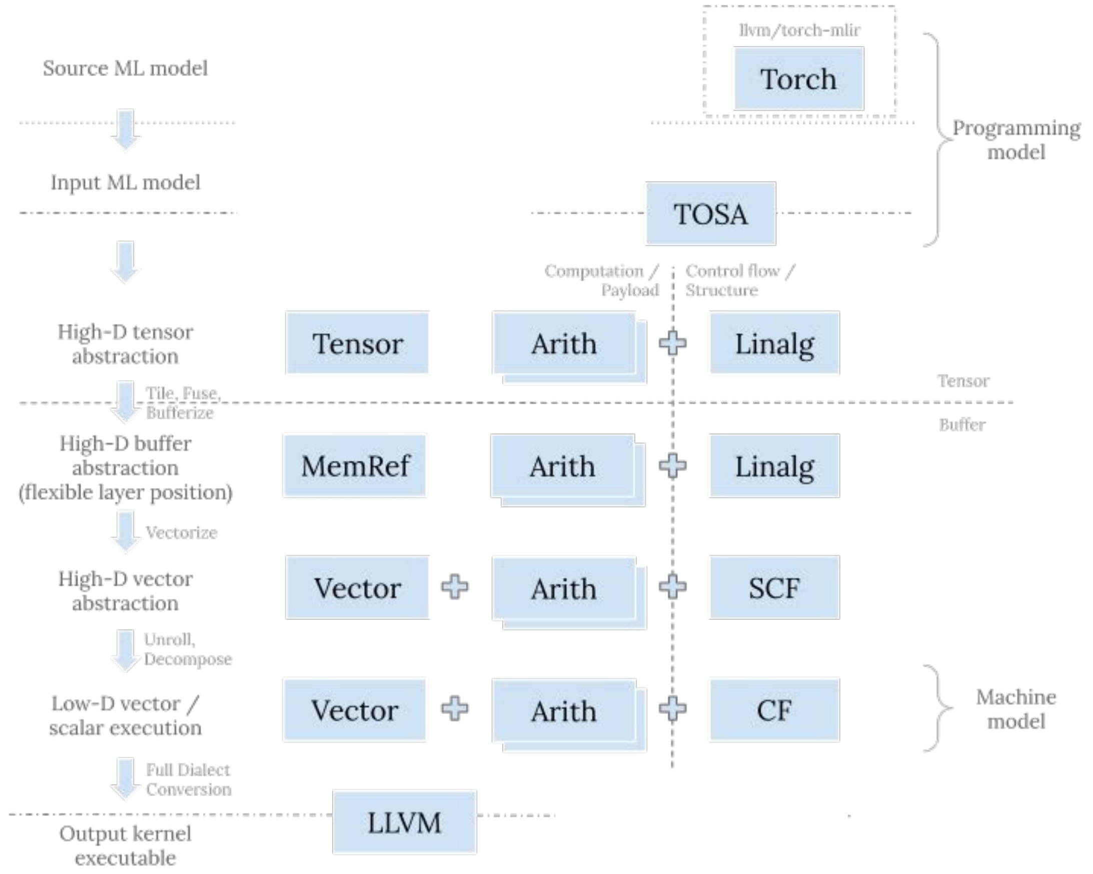
ML Model Compilation: Lower to LLVM

Memref, Vector, Arith, and CF to LLVM Lowering

```
  ... ...
^bb1(%20: i64):  // 2 preds: ^bb0, ^bb4
  %21 = llvm.icmp "slt" %20, %5 : i64
  llvm.cond_br %21, ^bb2(%4 : i64), ^bb5
^bb2(%22: i64):  // 2 preds: ^bb1, ^bb3
  %23 = llvm.icmp "slt" %22, %7 : i64
  llvm.cond_br %23, ^bb3, ^bb4
^bb3:  // pred: ^bb2
  ... ...
  %46 = llvm.intr.masked.load %45, %36, %0 {alignment = 4 : i32} : (!llvm.ptr<vector<2xf32>>, vector<2xi1>, vector<2xf32>)
-> vector<2xf32>
  %47 = llvm.fmul %30, %41  : vector<2xf32>
  %48 = llvm.fadd %46, %47  : vector<2xf32>
  llvm.intr.masked.store %48, %45, %36 {alignment = 4 : i32} : vector<2xf32>, vector<2xi1> into !llvm.ptr<vector<2xf32>>
  %49 = llvm.add %22, %8  : i64
  llvm.br ^bb2(%49 : i64)
^bb4:  // pred: ^bb2
  %50 = llvm.add %20, %6  : i64
  llvm.br ^bb1(%50 : i64)
^bb5:  // pred: ^bb1
  llvm.return
}
```

LLVM Dialect

MLIR Dialects Walk-through: Recall



Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, L. Zhang.

MLIR/LLVM Community

- Website: https://mlir.llvm.org/
- GitHub: https://github.com/llvm/llvm-project/tree/main/mlir
- Forums: https://llvm.discourse.group/c/mlir/31
- Discord: https://discord.gg/xS7Z362
- Youtube: https://www.youtube.com/MLIRCompiler

Outline

- **Background:** From LLVM to MLIR
- **MLIR:** Multi-Level Intermediate Representation
- **CIRCT:** Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

CIRCT: Compiler Infrastructure for the future of EDA

- **C**ircuit **I**ntermediate **R**epresentation **C**ompilers and **T**ools
- Built using MLIR
- LLVM incubator project
- **Composable toolchain for different aspects of electronic design automation (EDA) process**
- Common platform with clean interfaces
- Tools for designing accelerators are relevant for programming accelerators

https://circt.llvm.org

Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Parse Chisel Design into MLIR

```
module Foo:
  input clk: Clock
  input bus: {valid: UInt<1>, data: UInt<32>}

  reg dataReg: UInt, clk

  when bus.valid:
    dataReg <= bus.data
```

.fir file from Chisel

FIRRTL Parsing

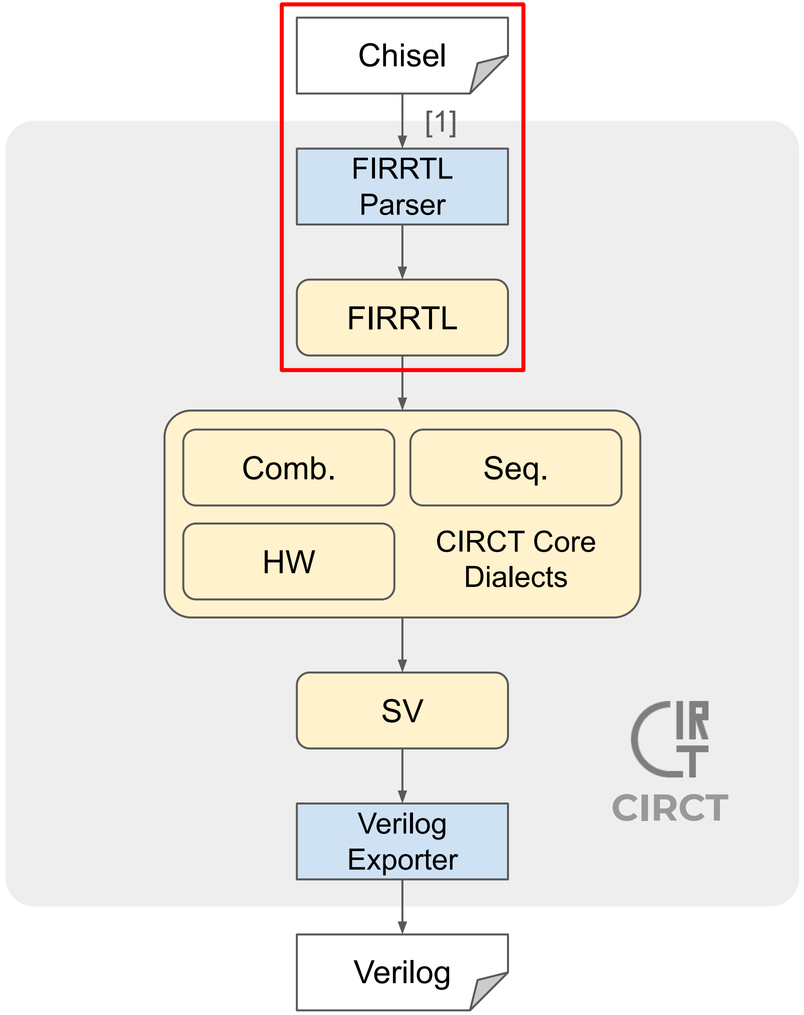
```
firrtl.module @Foo(in %clk: !firrtl.clock, in %bus:
                   !firrtl.bundle<valid: uint<1>, data: uint<32>>) {
  %dataReg = firrtl.reg %clk  : (!firrtl.clock) -> !firrtl.uint

  %0 = firrtl.subfield %bus("valid") :
       (!firrtl.bundle<valid: uint<1>, data: uint<32>>) -> !firrtl.uint<1>

  firrtl.when %0  {
    %1 = firrtl.subfield %bus("data") :
         (!firrtl.bundle<valid: uint<1>, data: uint<32>>) -> !firrtl.uint<32>

    firrtl.connect %dataReg, %1 : !firrtl.uint, !firrtl.uint<32>
} }
```

.mlir file



Circuit Transform in FIRRTL Dialect

```
firrtl.module @Foo(in %clk: !firrtl.clock, in %bus:
                   !firrtl.bundle<valid: uint<1>, data: uint<32>>) {
  %dataReg = firrtl.reg %clk  : (!firrtl.clock) -> !firrtl.uint

  %0 = firrtl.subfield %bus("valid") :
       (!firrtl.bundle<valid: uint<1>, data: uint<32>>) -> !firrtl.uint<1>

  firrtl.when %0  {
    %1 = firrtl.subfield %bus("data") :
         (!firrtl.bundle<valid: uint<1>, data: uint<32>>) -> !firrtl.uint<32>

    firrtl.connect %dataReg, %1 : !firrtl.uint, !firrtl.uint<32>
} }
```

.mlir file: High FIRRTL

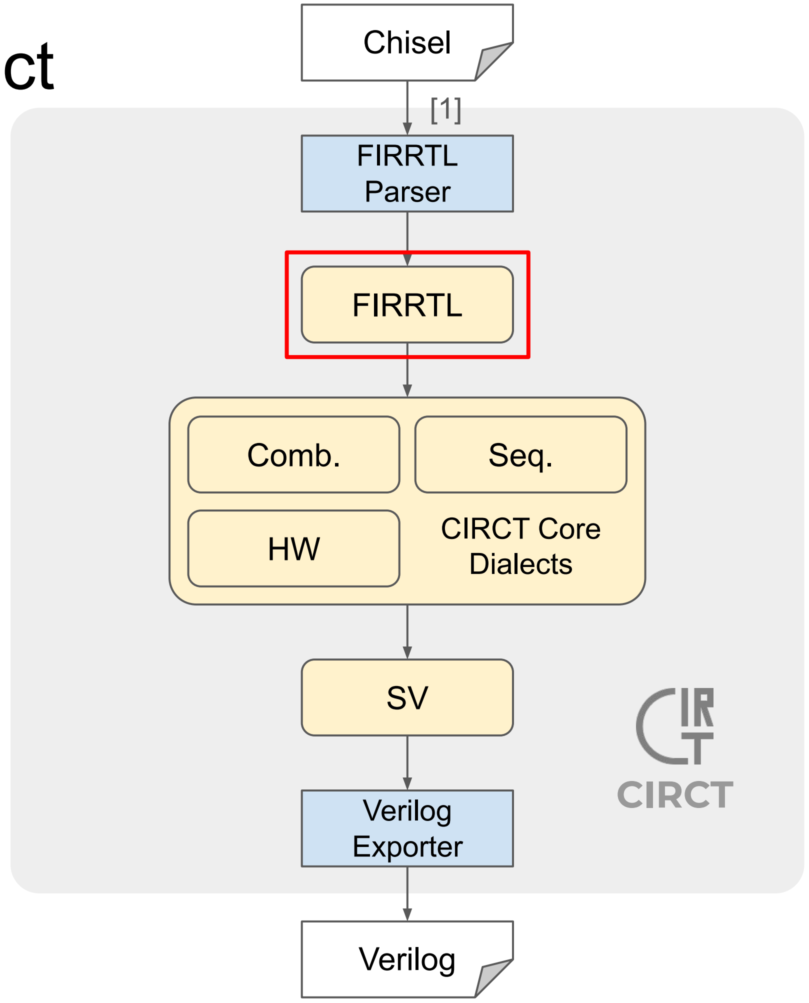
High FIRRTL to Low FIRRTL

```
firrtl.module @Foo(in %clk: !firrtl.clock, in %bus_valid: !firrtl.uint<1>,
                   in %bus_data: !firrtl.uint<32>) {
  %dataReg = firrtl.reg %clk  : (!firrtl.clock) -> !firrtl.uint<32>

  %0 = firrtl.mux(%bus_valid, %bus_data, %dataReg) :
    (!firrtl.uint<1>, !firrtl.uint<32>, !firrtl.uint<32>) -> !firrtl.uint<32>

  firrtl.connect %dataReg, %0 : !firrtl.uint<32>, !firrtl.uint<32>
}
```

.mlir file: Low FIRRTL

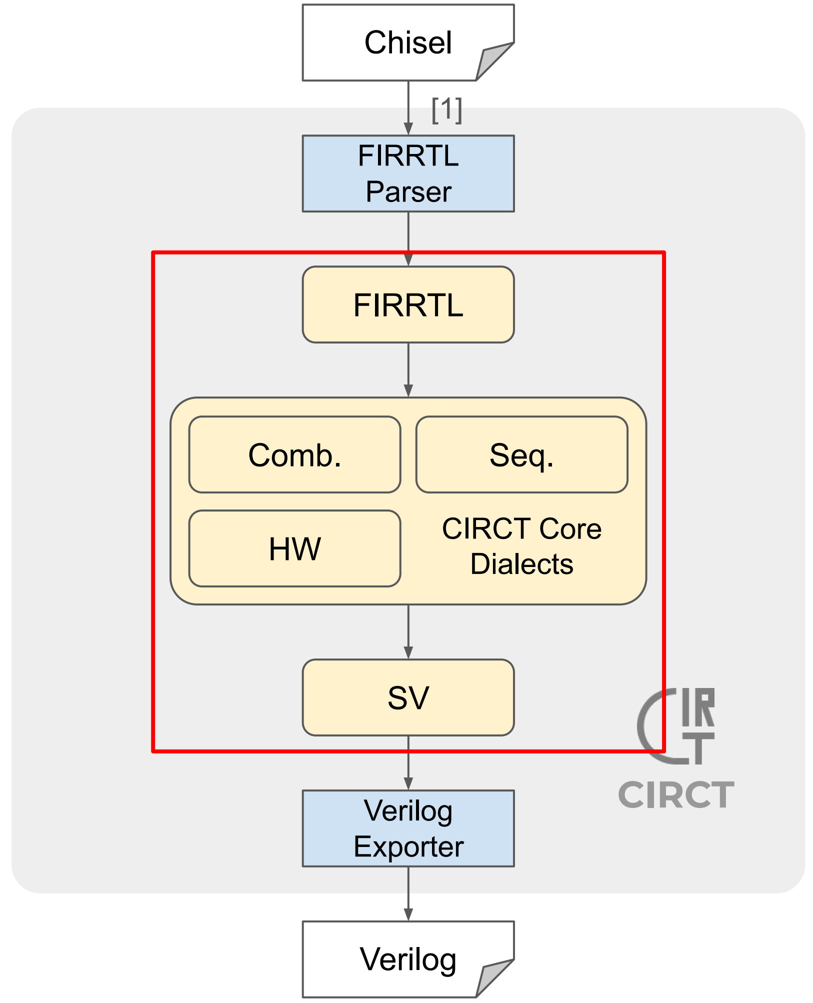


Lower to CIRCT Core Dialects

FIRRTL to HW Lowering

```
hw.module @Foo(%clk: i1, %bus_valid: i1, %bus_data: i32) {
  %dataReg = sv.reg : !hw.inout<i32>
  sv.ifdef "SYNTHESIS"  {
  } else  {
    sv.initial  {
      sv.verbatim "`INIT_RANDOM_PROLOG_"
      sv.ifdef.procedural "RANDOMIZE_REG_INIT"  {
        %RANDOM = sv.verbatim.expr "`RANDOM" : () -> i32
        sv.bpassign %dataReg, %RANDOM : i32
      }
    }
  }
  %0 = sv.read_inout %dataReg : !hw.inout<i32>
  %1 = comb.mux %bus_valid, %bus_data, %0 : i32
  sv.alwaysff(posedge %clk)  {
    sv.passign %dataReg, %1 : i32
  }
  hw.output
}
```

.mlir file: HW+Comb+SV



[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

Export the IR as SystemVerilog

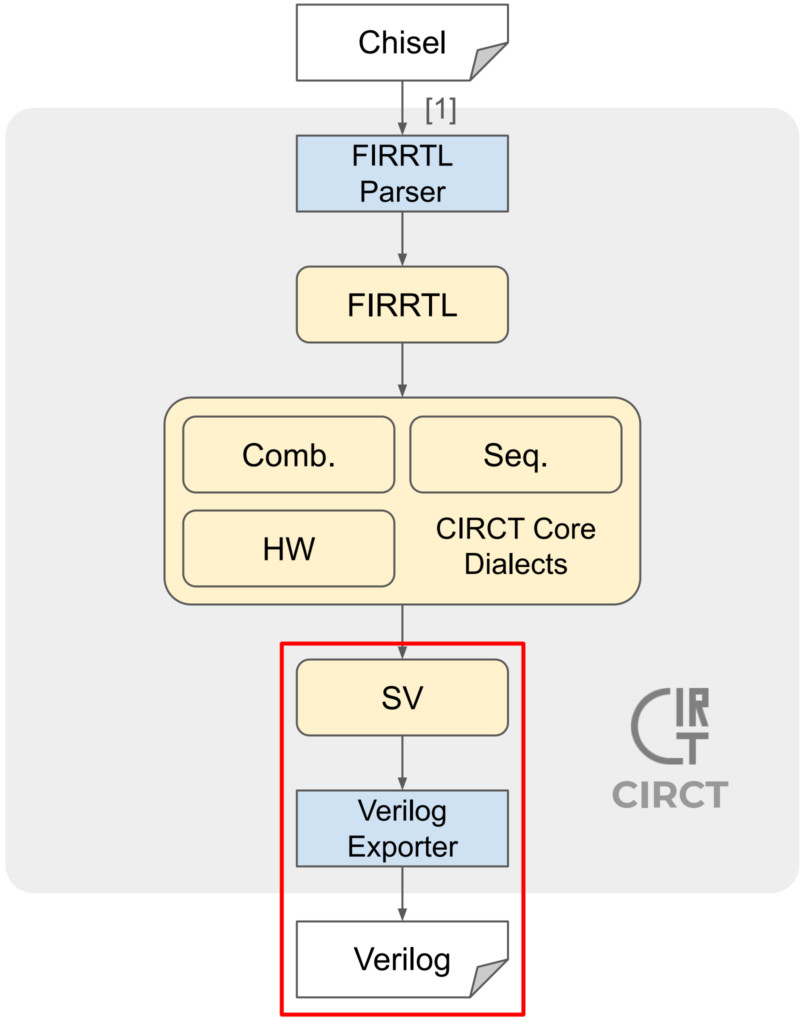
Export SystemVerilog

```
module Foo(
 input        clk, bus_valid,
 input [31:0] bus_data);

 reg [31:0] dataReg;   // Foo.mlir:32:16

 `ifndef SYNTHESIS     // Foo.mlir:33:5
   initial begin       // Foo.mlir:35:7
     `INIT_RANDOM_PROLOG_      // Foo.mlir:36:9
     `ifdef RANDOMIZE_REG_INIT // Foo.mlir:37:9
      dataReg = `RANDOM;       // Foo.mlir:38:21, :39:11
     `endif
   end // initial
 `endif
 wire [31:0] _T = bus_valid ? bus_data : dataReg;
// Foo.mlir:43:10, :44:10
 always_ff @(posedge clk)       // Foo.mlir:45:5
   dataReg <= _T;       // Foo.mlir:46:7
endmodule
```

.sv file



[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

Represent Circuits: Core Dialects

HW Dialect

- Abstract the structure of hardware circuits, such as *(Ext)Module/Instance*, and types, such as *InOut, Array, Struct, Union, etc.*
- Module port can support different types, such as SystemVerilog Interface, in order to abstract hardware circuits at different abstractions.
- Can combine with dialects apart from Comb and Seq.
- Convenient for IR analysis and transform.

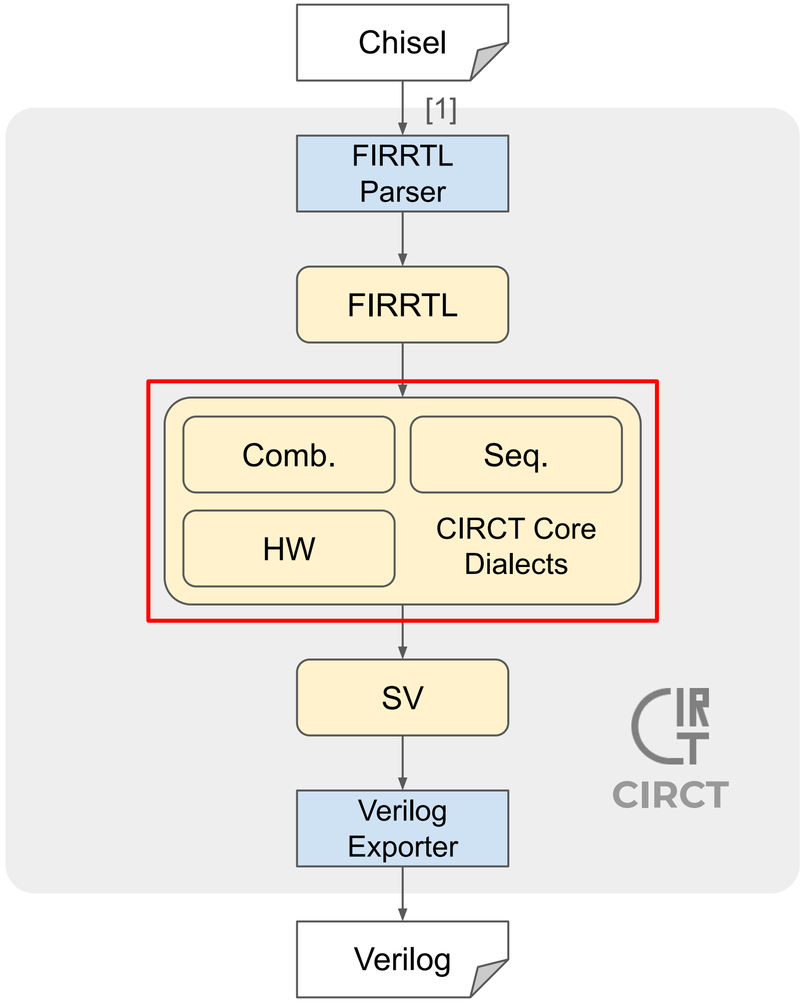
Comb and Seq Dialect

- Represent combinational and sequential logics.

SV Dialect

- Represent declarations and structures in SystemVerilog in order to print pretty .sv file.

[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

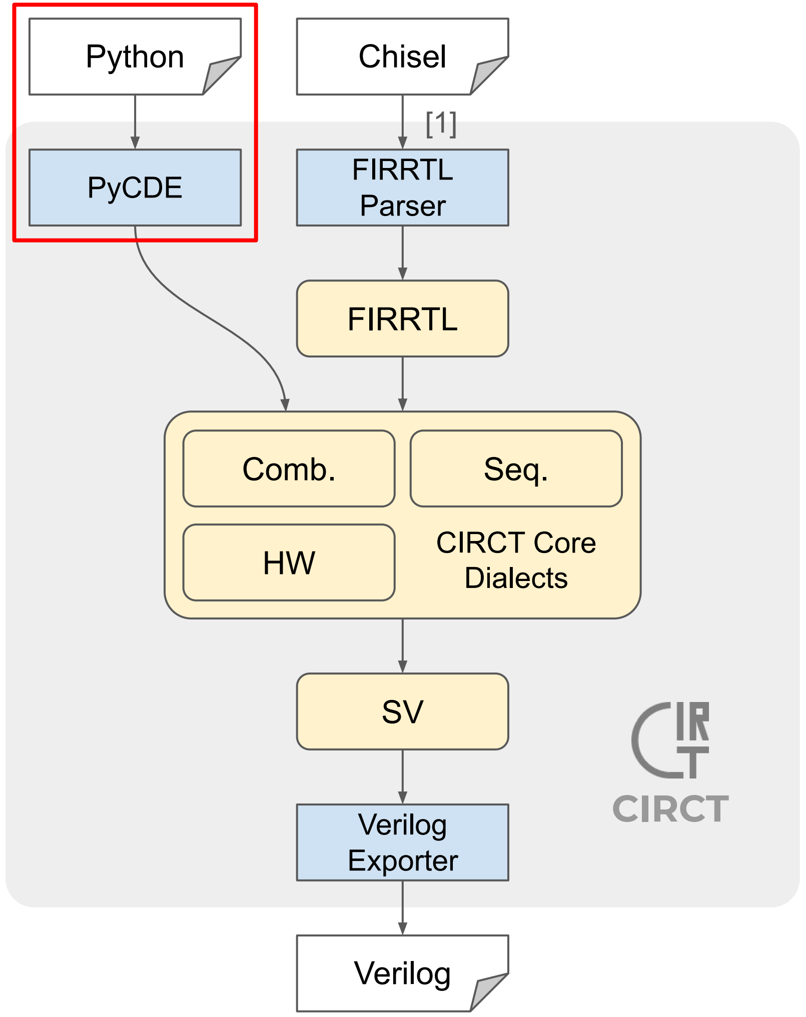


Modular & Extensible: PyCDE

Python CIRCT Design Entry (PyCDE)

- Meta HDL based on Python language.
- Parse into MLIR through Python binding.
- Can reuse core dialects for circuit optimization.
- Can reuse the SV dialect and Verilog exporter for pretty verilog generation.

CIRCT can boost the design and implementation of hardware programming language



[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

Modular & Extensible: Simulation

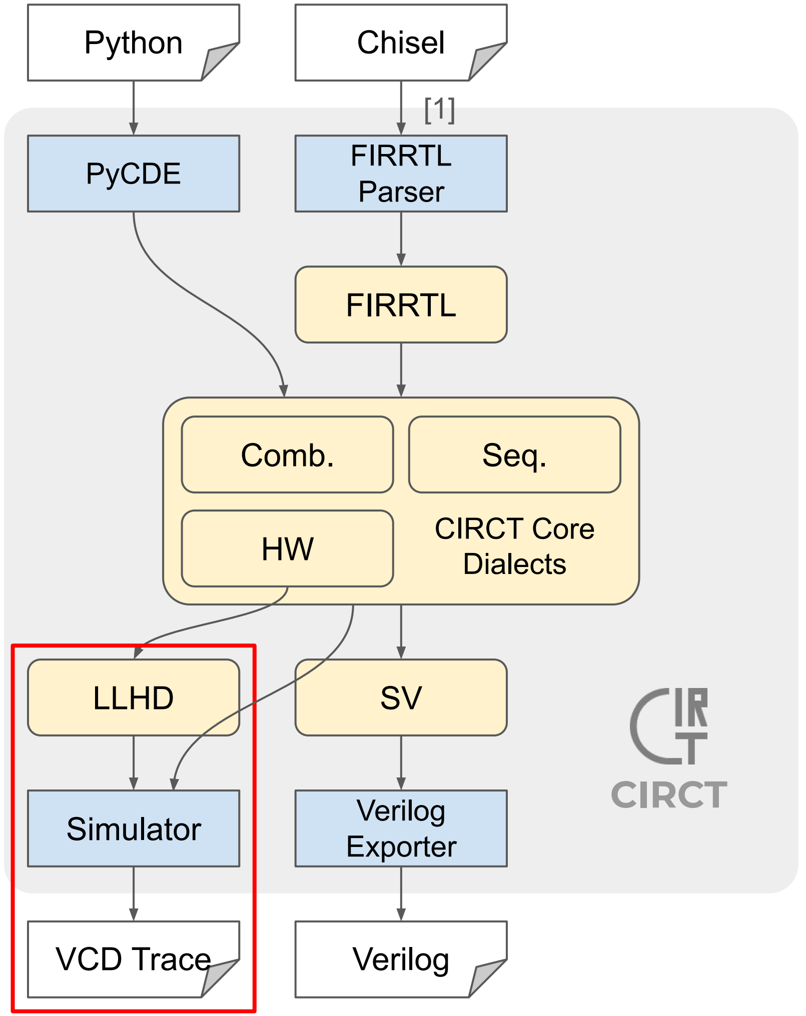
LLHD (Low Level Hardware Description) Dialect

- Dedicated for low-level circuit representation
- Explicitly carry timing information
- Support MLIR-based circuit simulation

⇩ *ongoing*

- Support massive parallelization in the simulation

CIRCT can boost the hardware simulation techniques

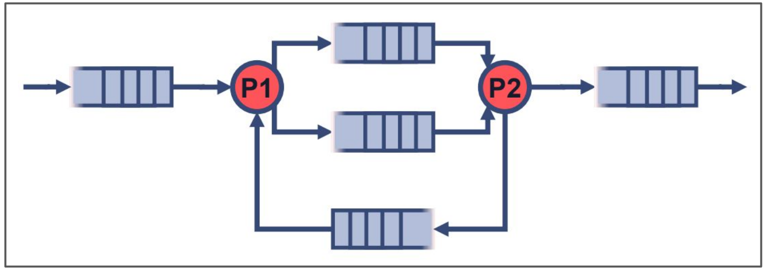


[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

Modular & Extensible: HLS

Handshake Dialect

- Processes communicate through stream interfaces.
- Interfaces connected by single-reader single-writer FIFOs, which are logically unbounded.
- Processes can access interfaces in any order.
- Provably deterministic if processes cannot test state of streams: *Elastic* and *Latency Insensitive*

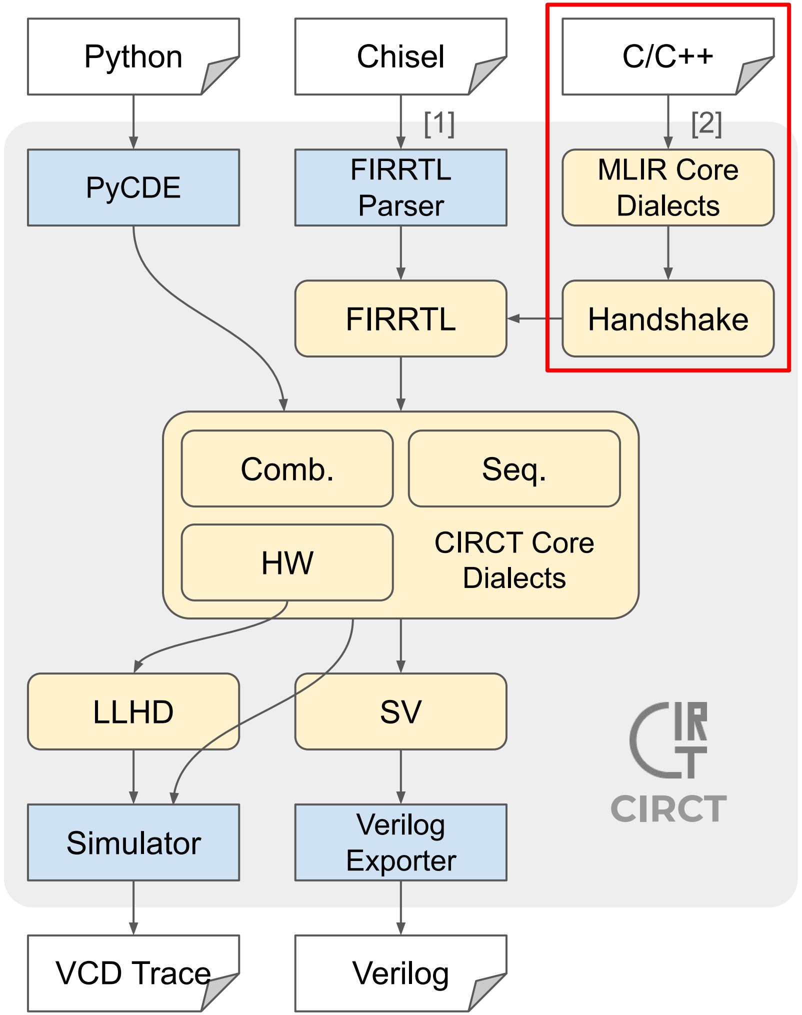


CIRCT can boost High-level Synthesis research

Source: Handshake-based HLS in CIRCT, H. Ye.

[1] Chisel3: https://github.com/chipsalliance/chisel3

[2] Polygeist: https://github.com/wsmoses/Polygeist



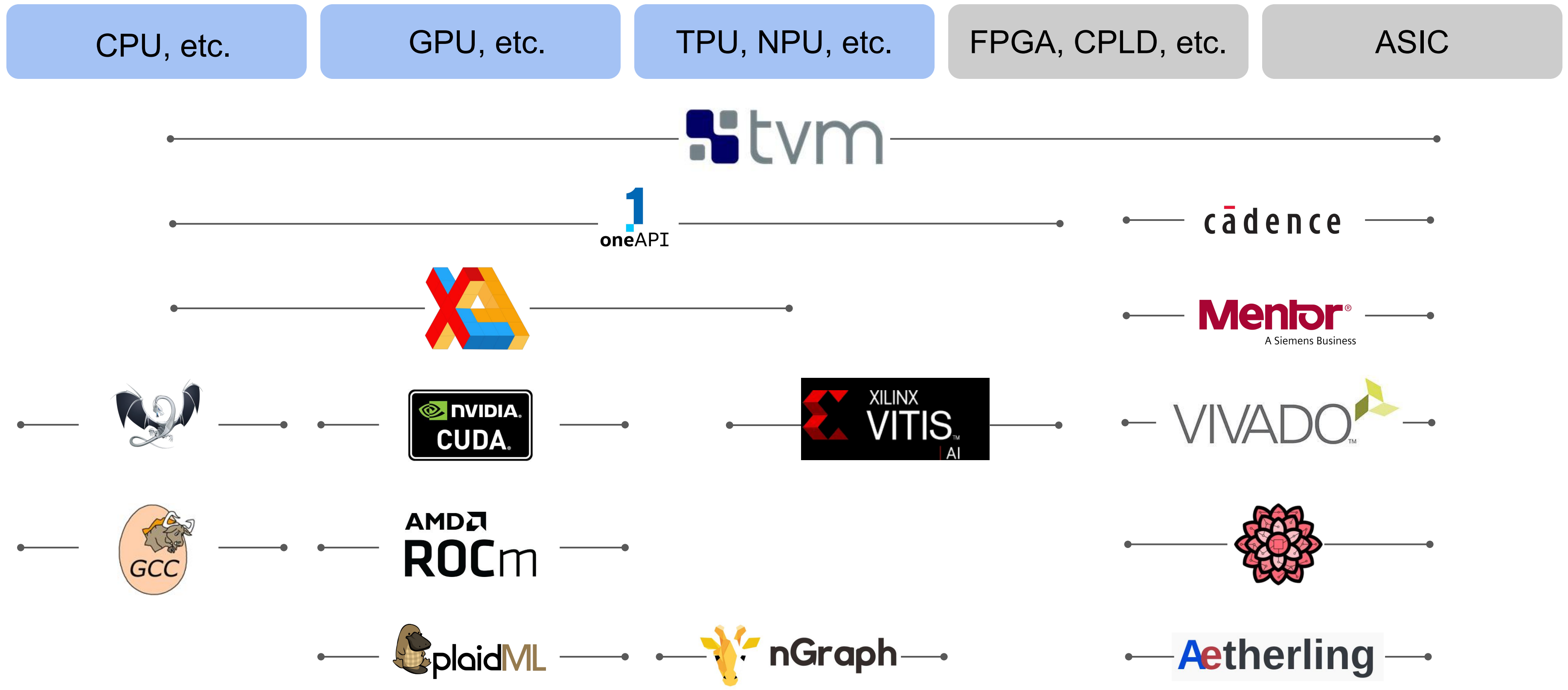
CIRCT Community

- Website: https://circt.llvm.org/
- GitHub: https://github.com/llvm/circt/tree/main/
- Forums: https://llvm.discourse.group/c/Projects-that-want-to-become-official-LLVM-Projects/circt/
- Discord: https://discord.com/channels/636084430946959380/742572728787402763

Outline

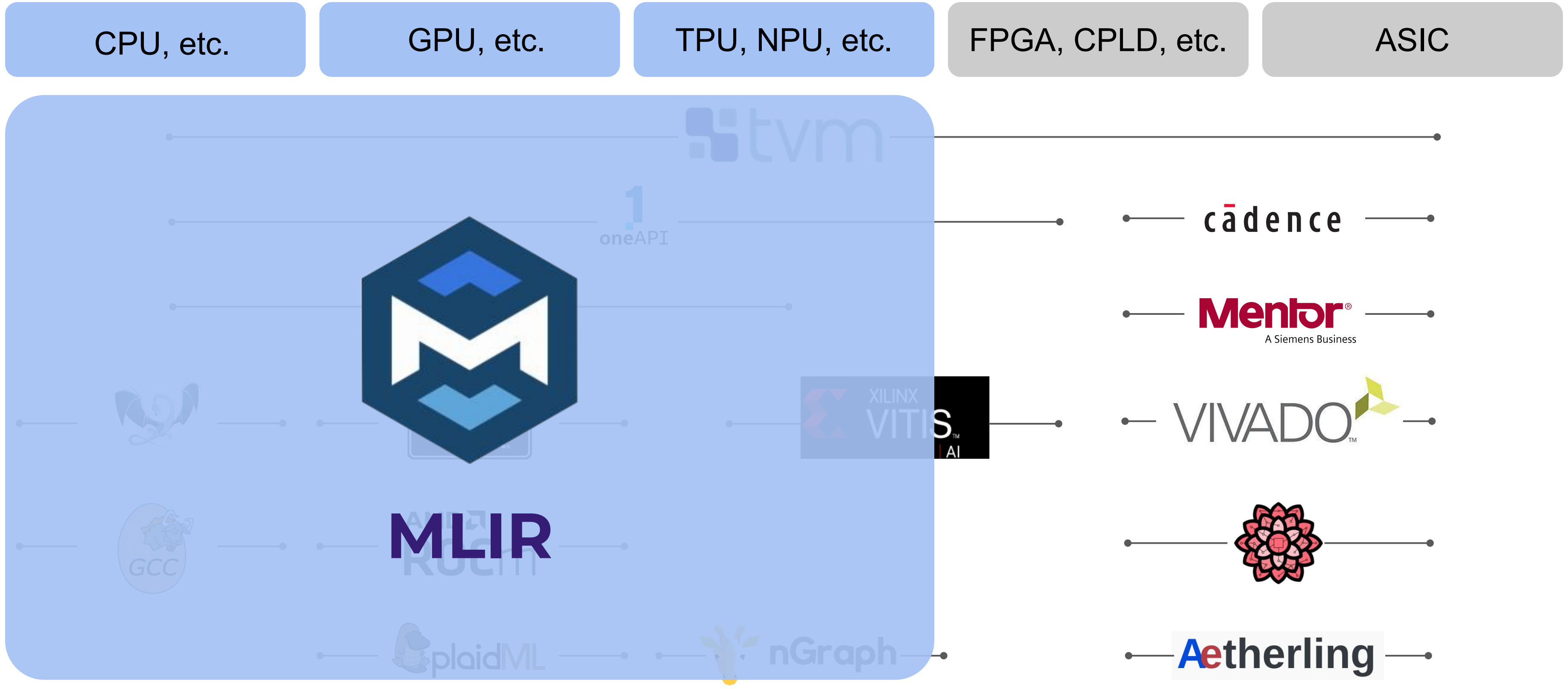
- **Background:** From LLVM to MLIR
- **MLIR:** Multi-Level Intermediate Representation
- **CIRCT:** Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

Spectrum of Compilers



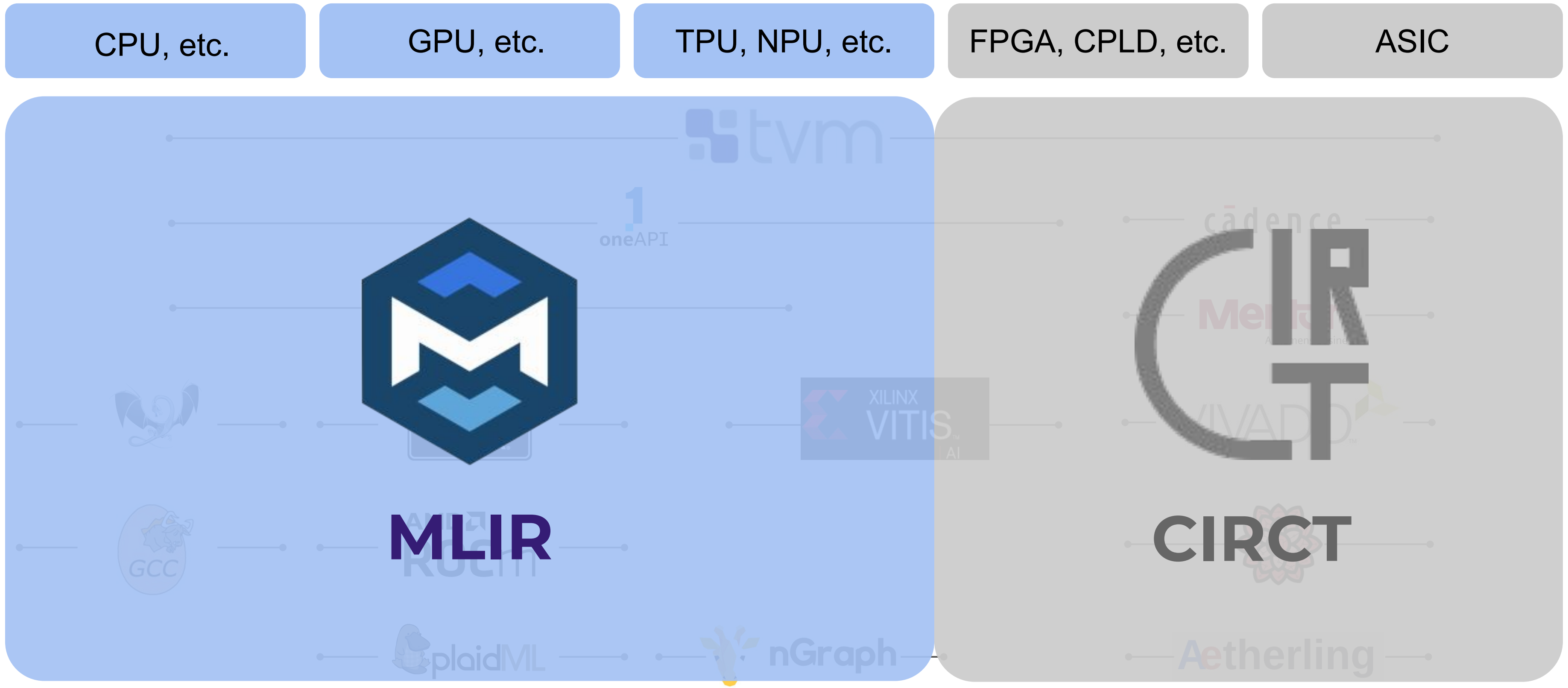
Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Spectrum of Compilers (Cont'd)



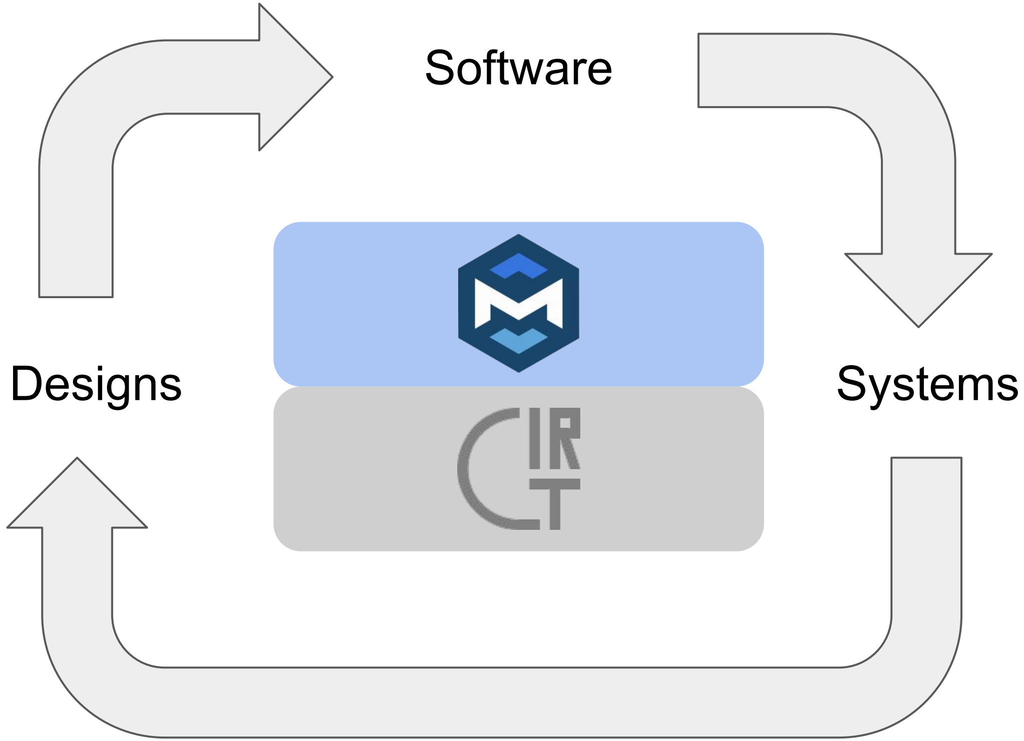
Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Spectrum of Compilers (Cont’d)



Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Hardware and Software Co-design



Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Thanks!
Q&A

Hanchen Ye
hanchenye@gmail.com
July 28, 2022