Compilers for Domain-Specific Accelerators

Hanchen Ye
hanchenye@gmail.com
Dec. 2, 2021

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 research lies in the area of high-level synthesis (HLS), domain-specific compilers, and hardware acceleration. He has published multiple conference papers on DAC, ICCAD, HPCA, etc. He has been regularly contributing to open-source projects, including MLIR, CIRCT, MLIR-AIE, etc.

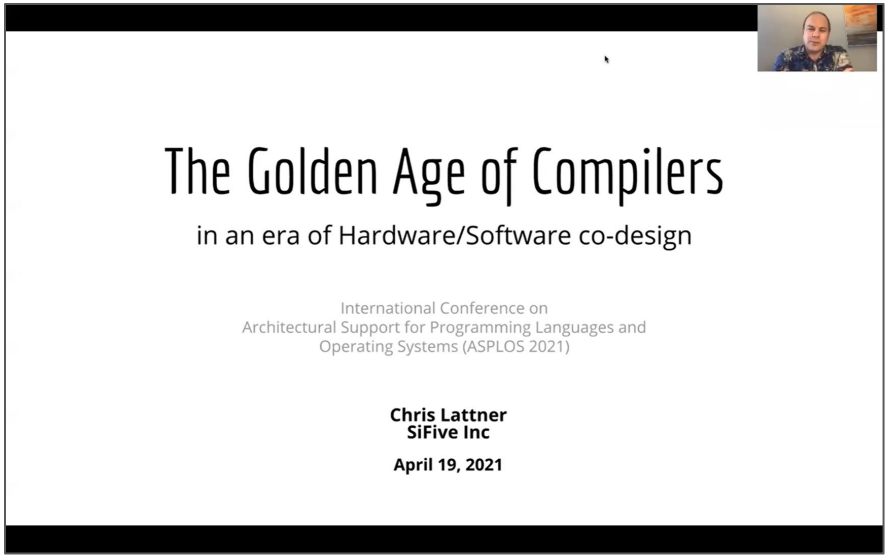
Outline

- **Background:** Domain-Specific Accelerator (DSA)
- **Software Compilation:** How to program DSA?
  - MLIR: Multi-Level Intermediate Representation
- **Hardware Compilation:** How to design and verify DSA?
  - CIRCT: Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

Outline

- **Background:** Domain-Specific Accelerator (DSA)
- **Software Compilation:** How to program DSA?
  - MLIR: Multi-Level Intermediate Representation
- **Hardware Compilation:** How to design and verify DSA?
  - CIRCT: Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

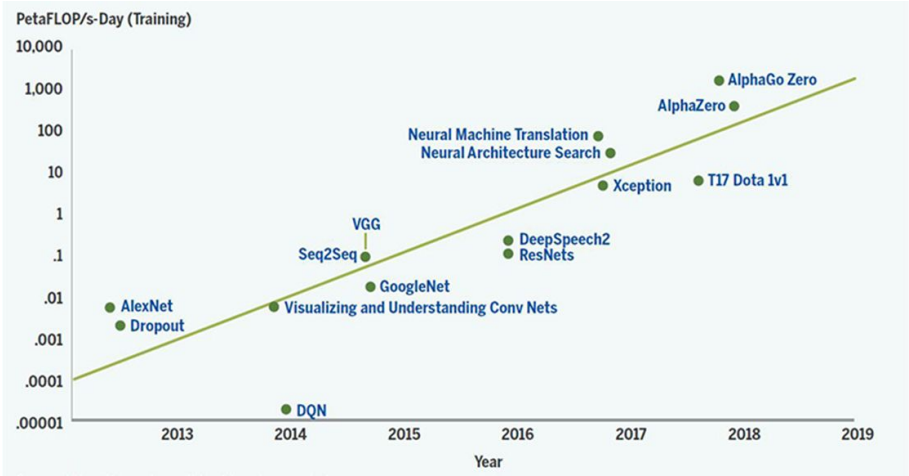
The Golden Age of Architecture and Compiler



- A New Golden Age for Computer Architecture, David Patterson, 2019.
- The Golden Age of Compilers, Chris Lattner, 2021.

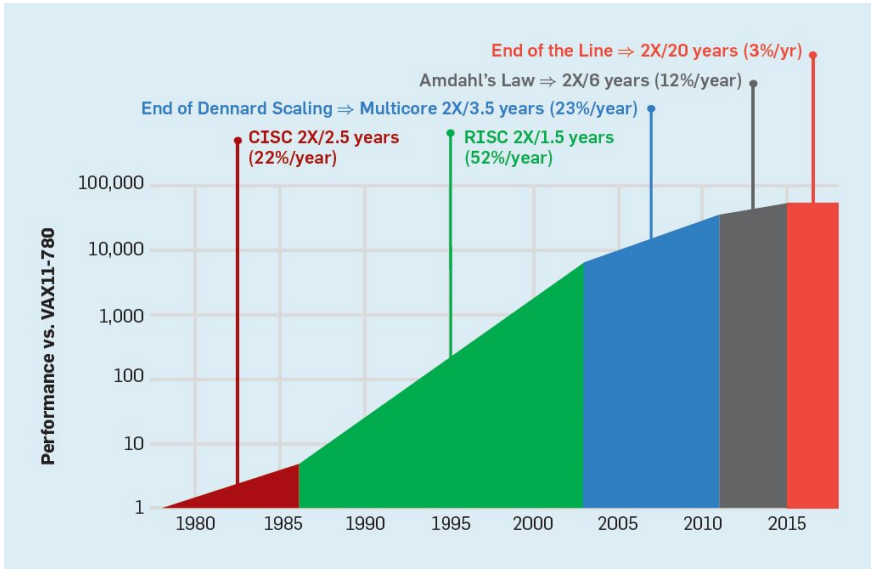
Background of Domain-Specific Accelerator (DSA)

Exponentially increasing computing demands



Source: AI and Compute, D. Amodei and D. Hernandez

The End of Moore's Law



Source: A New Golden Age for Computer Architecture, J. Hennessy and D. Patterson

Opportunities and Challenges of DSA

What Opportunities Left? (Part II)

- Only performance path left is **Domain Specific Architectures (DSAs)**
  - Just do a few tasks, but extremely well
- Achieve higher efficiency by tailoring the architecture to characteristics of the domain
- Not one application, but a domain of applications
- Different from strict ASIC since still runs software

28

Why DSAs Can Win (no magic)
Tailor the Architecture to the Domain

- More effective parallelism for a specific domain:
  - SIMD vs. MIMD
  - VLIW vs. Speculative, out-of-order
- More effective use of memory bandwidth
  - User controlled versus caches
- Eliminate unneeded accuracy
  - IEEE replaced by lower precision FP
  - 32-64 bit integers to 8-16 bit integers
- Domain specific programming language provides path for software

29

Source: A New Golden Age for Computer Architecture, J. Hennessy and D. Patterson

Opportunities and Challenges of DSA (Cont'd)

Hardware is getting **hard**er

Modern compute acceleration platforms are multi-level and explicit:

- Scalar, SIMD/Vector, Multi-core, Multi-package, Multi-rack
- Non-coherent memory subsystems increase efficiency

Heterogeneous compute incorporating domain-specific accelerators

- Standard in high-end SoCs, domain-specific hard blocks in FPGAs

Many accelerator IPs are configurable:

- Optional extensions, tile / core count, memory hierarchy, etc

How can "normal people" write Software for this in the first place?
... and how can you *afford* to build generation-specific SW?

Source: The Golden Age of Compilers, C. Lattner.

Outline

- **Background:** Domain-Specific Architecture (DSA)
- **Software Compilation:** How to program DSA?
  - MLIR: Multi-Level Intermediate Representation
- **Hardware Compilation:** How to design and verify DSA?
  - CIRCT: Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

How to program DSA? Take AI DSA as example

Programming Languages

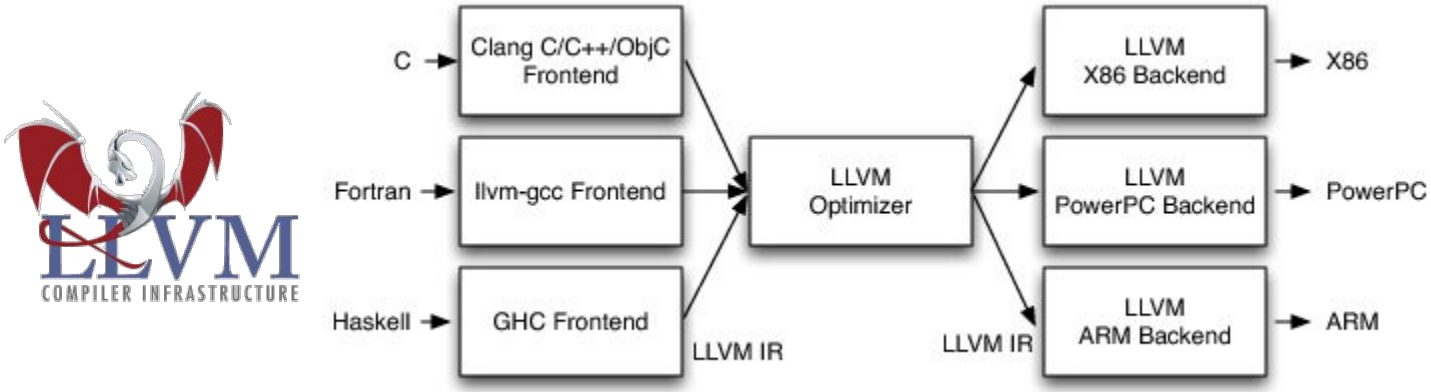
Domain-specific languages (DSL) or domain-specific programming frameworks

Compilers

1. Using traditional compilers, such as LLVM, for optimization and code generation?
   - LLVM is designed for CPU compilation and only supports low-level abstraction of programs.
2. Developing chip-specific compilation framework?
   - Instruction parallelization, multi-thread management, memory management, heterogeneous back-ends, code debugging, code generation, etc.

We need a **modular, extensible, and multi-level** compilation framework, which can be extended for the representation, optimization, and code generation of different domains. **MLIR!**

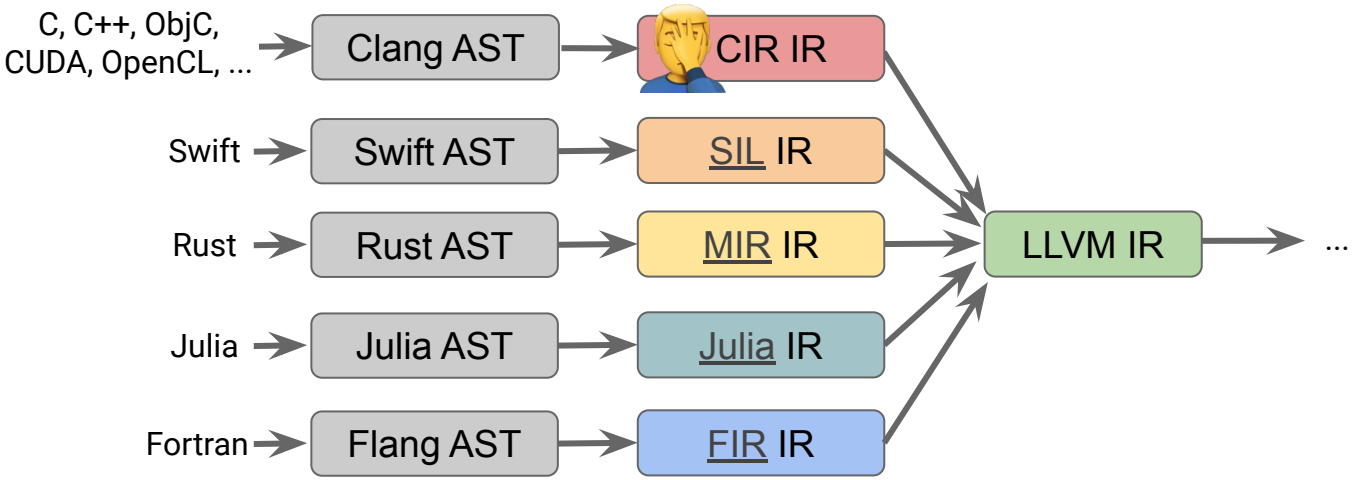
From LLVM to MLIR



- 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.

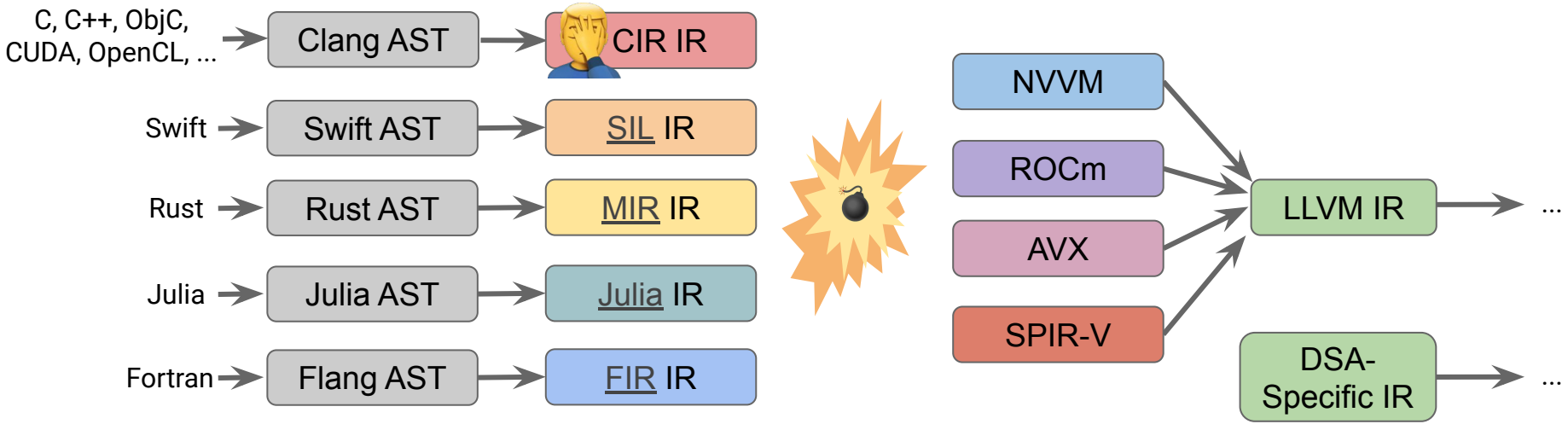
From LLVM to MLIR (Cont'd)



- 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

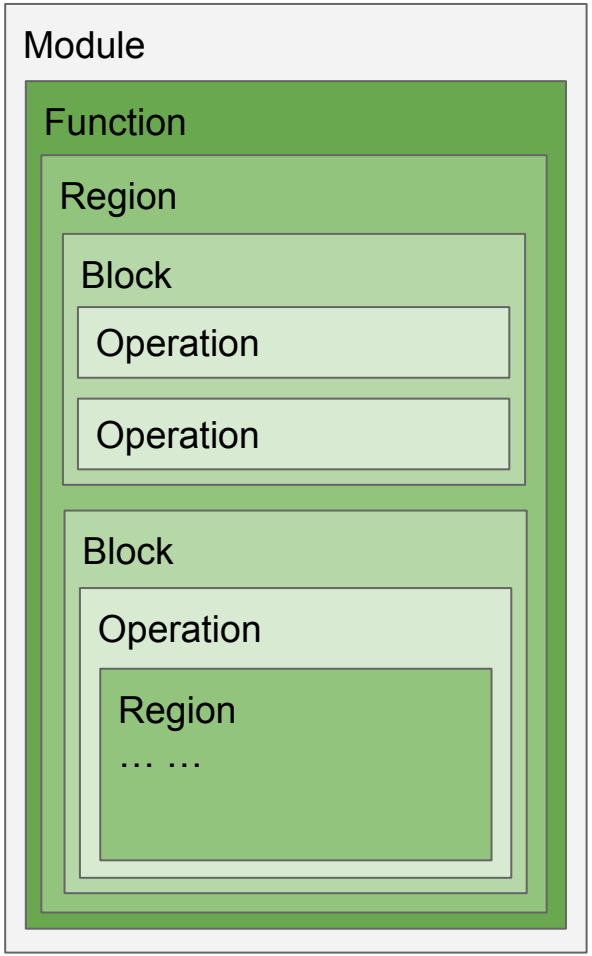
Source: Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications, M. Urbach.

Syntax of MLIR

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

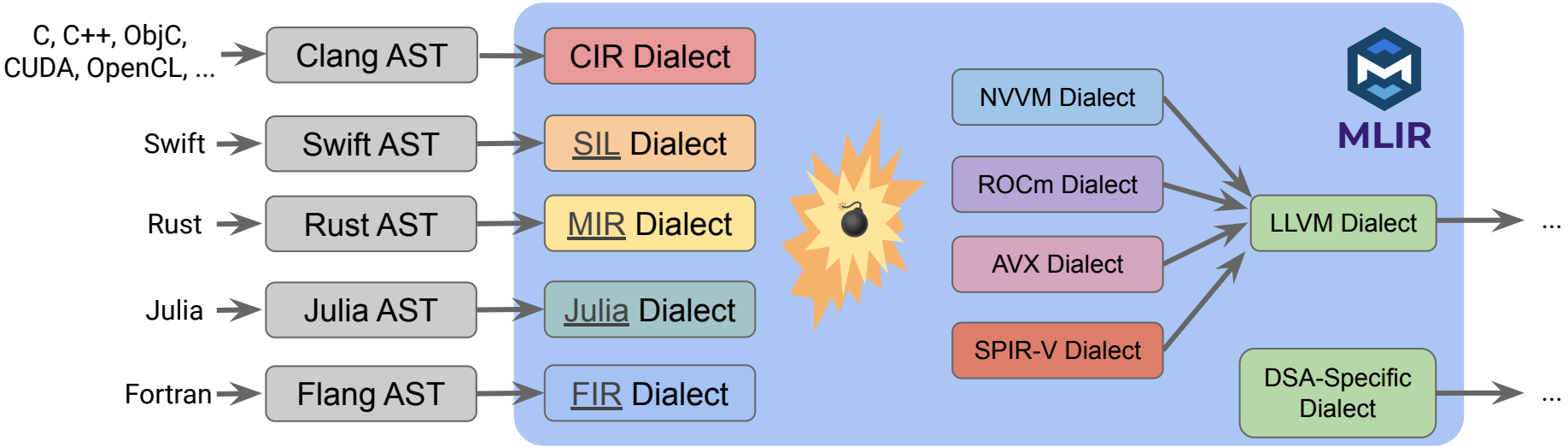
```
func @testFunction(%arg0: i32) {
  %x = call @thingToCall(%arg0) : (i32) -> i32
  br ^bb1
^bb1:
  %y = addi %x, %x : i32
  return %y : i32
}
```

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



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

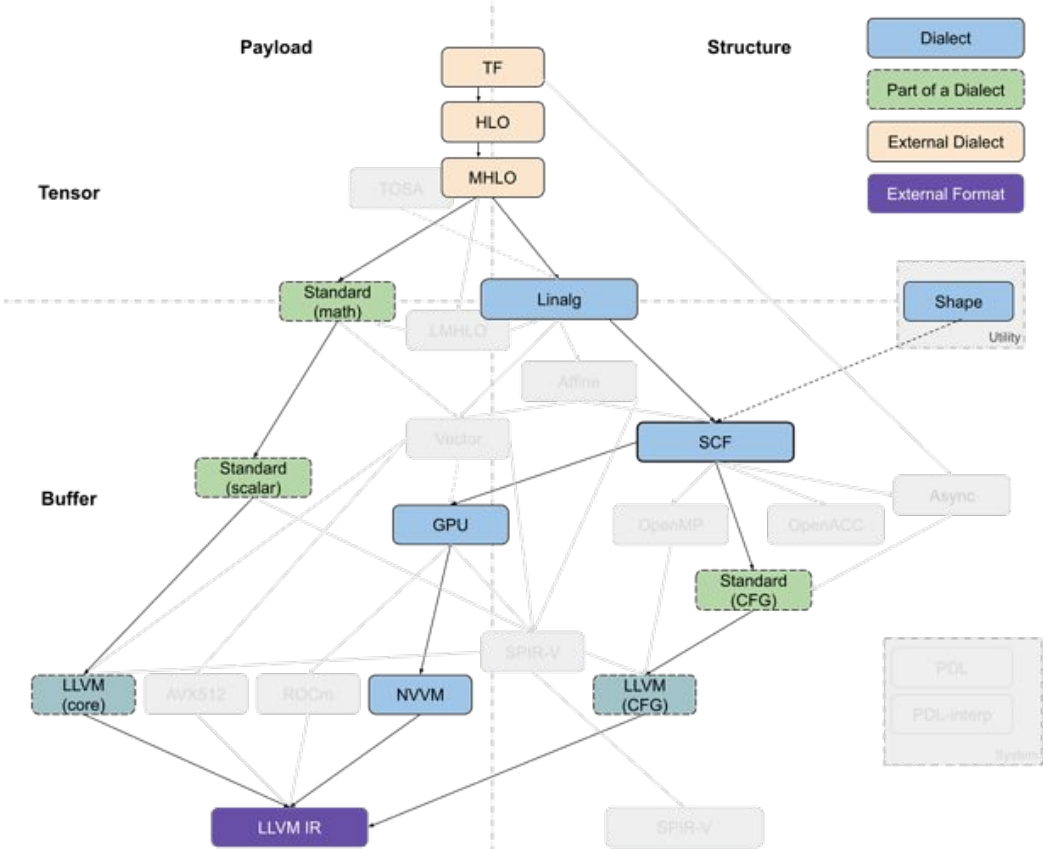
MLIR: “Meta IR” and Compiler Infrastructure



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**

MLIR-based Domain-Specific Compilation: Tensorflow



- **TF/HLO/MHLO:** Tensor level dialects representing the TF graphs
- **Linalg:** Linear algebra dialect
- **SCF (Structured Control Flow):** Loop level dialect explicitly representing loop structures
- **GPU:** Dialect dedicated for GPU-targeted optimizations
- **NVVM/LLVM:** Low-level dialect for CPU/GPU code generation

Source: https://llvm.discourse.group/t/codegen-dialect-overview/2723

Join the 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

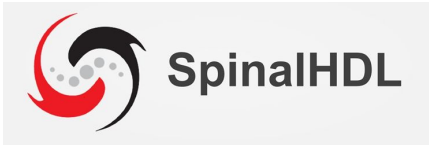
Outline

- **Background:** Domain-Specific Architecture (DSA)
- **Software Compilation:** How to program DSA?
  - MLIR: Multi-Level Intermediate Representation
- **Hardware Compilation:** How to design and verify DSA?
  - CIRCT: Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

How to design and verify DSA?

Design Languages

1. High-level Synthesis? Suitable for designing high-performance functional sub-modules.
2. Verilog/VHDL is industry standard, but: *Huge, compilated, incompletely implemented; Is it an IR? Or a programming language for humans?* [1]
3. Meta HDL? Chisel/SpinalHDL, CλaSH/Bluespec, and MyHDL/Migen generate Verilog from modern languages, such as Scala, Haskell, Python, etc.



[1] The golden age of compilers, C. Lattner.

EDA Tools (Compilers)

The optimization, synthesis, place & route, and verification can be implemented with compilation techniques:

1. Hardware circuit can be abstracted as IR, such as FIRRTL
2. Optimization can be implemented as the transform of IR, while synthesis and place & route as the lowering of IR
3. Verification can be implemented through IR analysis and simulation

We need modular and extensible **hardware compilation framework** to represent, optimize, and simulate hardware circuits.

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

circt-translate -import-firrtl

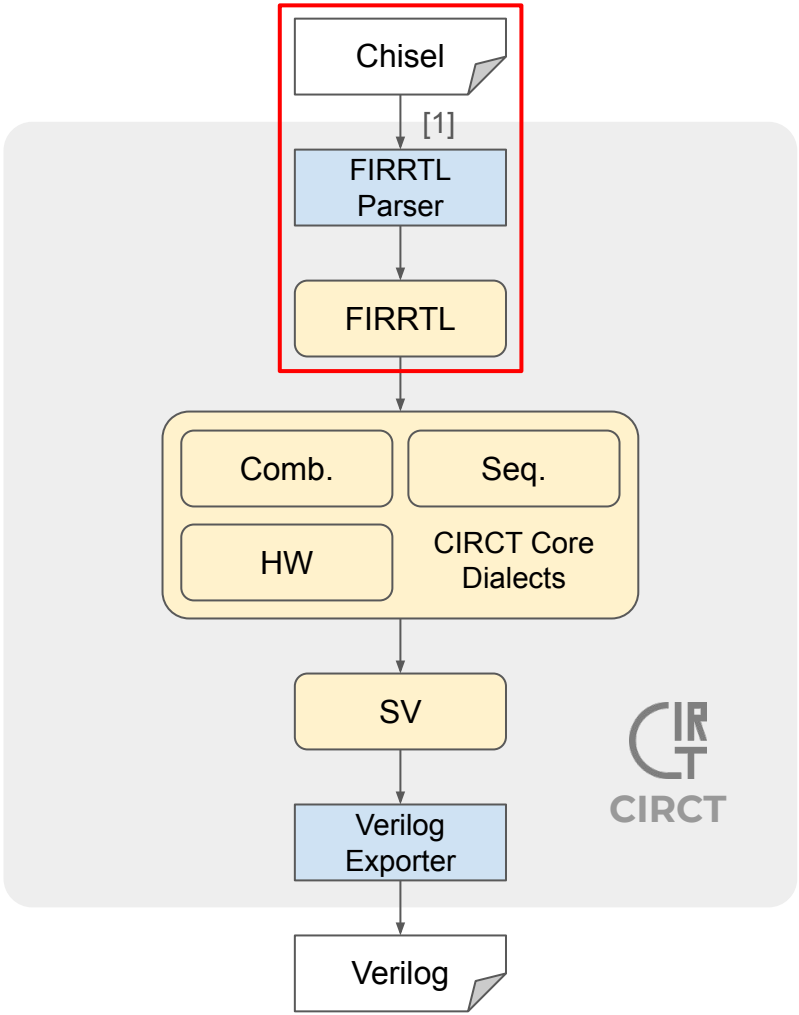
```
firrtl.module @Foo(in %clk: !firrtl.clock, in %bus:
                   !firrtl.bundle<valid: uint<1>, data: uint<32>>) {
  %dataReg = firrtl.reg %clk  : (!firrtl.clock) -> !firrtl.uint

  %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

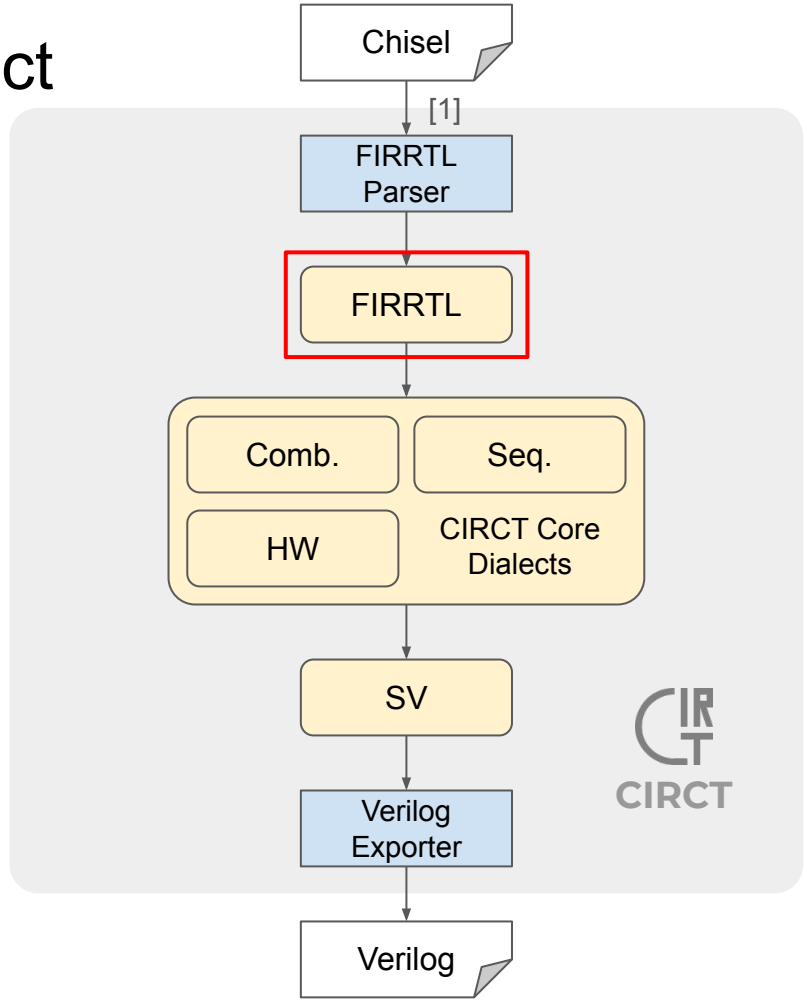
circt-opt -firrtl-lower-types -firrtl-infer-widths -firrtl-expand-whens

```
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

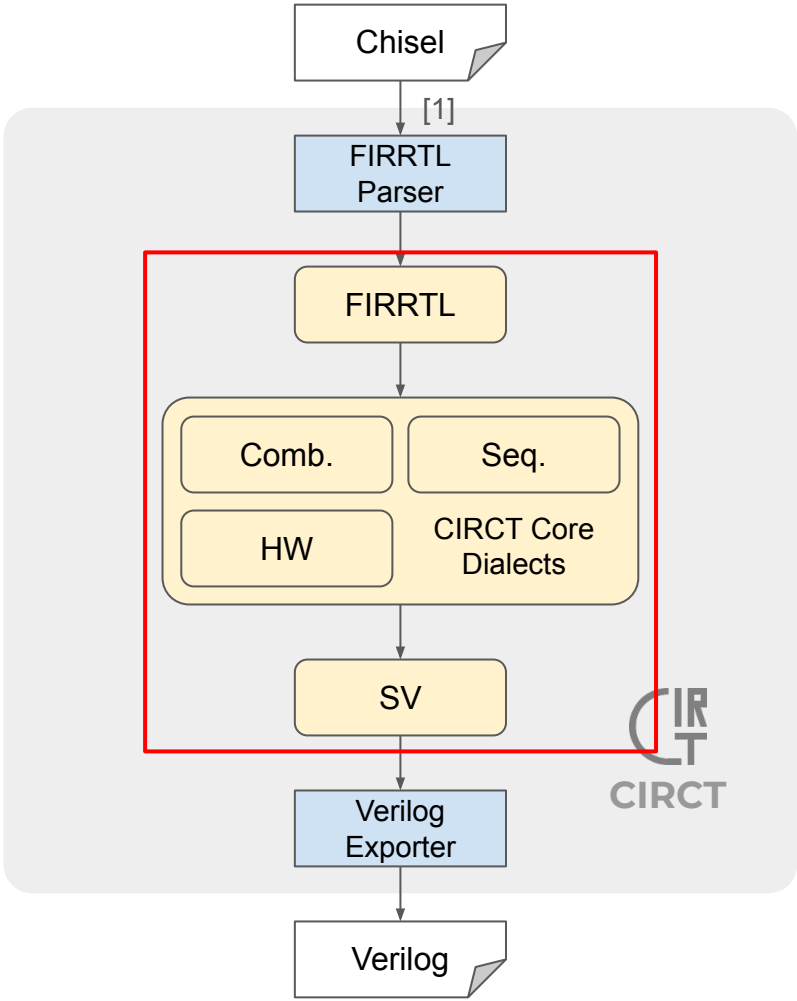


Lower to CIRCT Core Dialects

circt-opt -lower-firrtl-to-hw

```
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

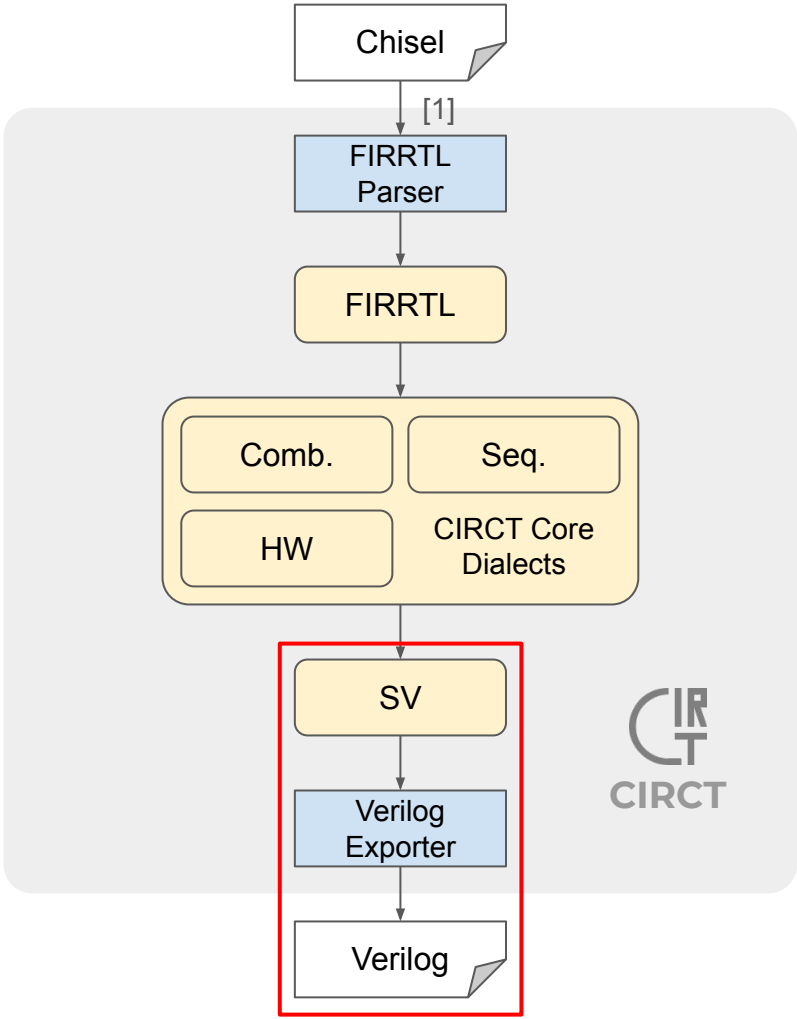
circt-translate -export-verilog

```
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

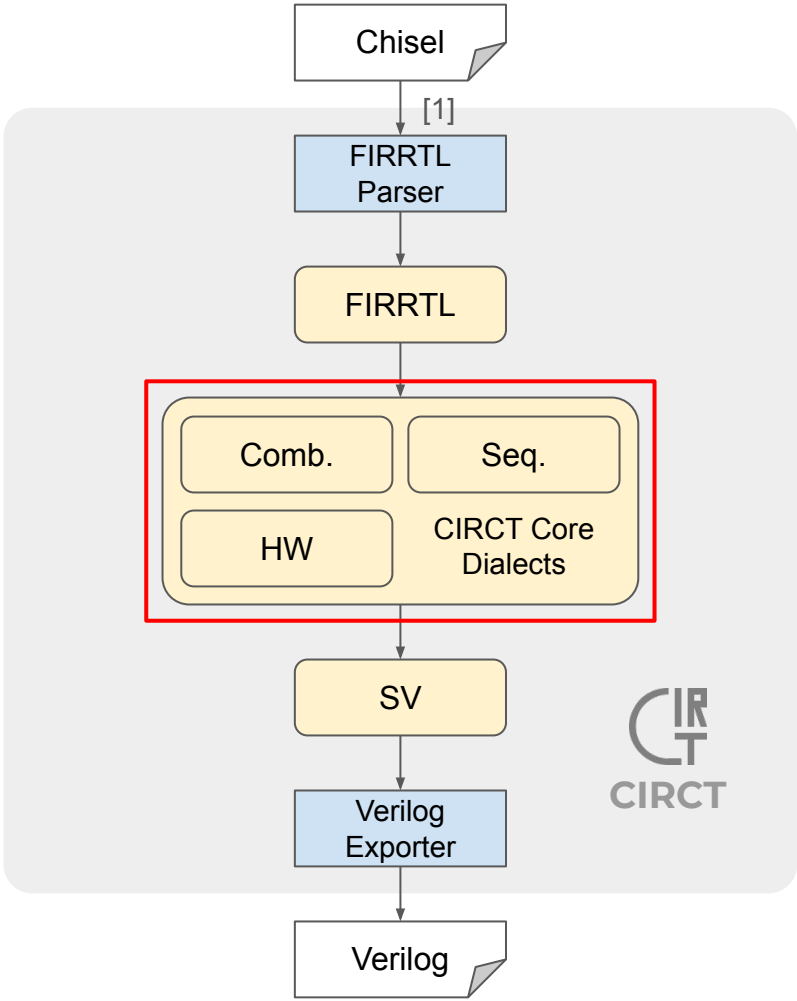
- Abstract the structure of hardware circuits, such as *(Ext)Module/Instance*, and types, such as *InOut, Array, Struct, Union, etc.*
- Module port can support different types, such as SystemVerilog Interface, in order to abstract hardware circuits at different abstractions.
- Can combine with dialects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



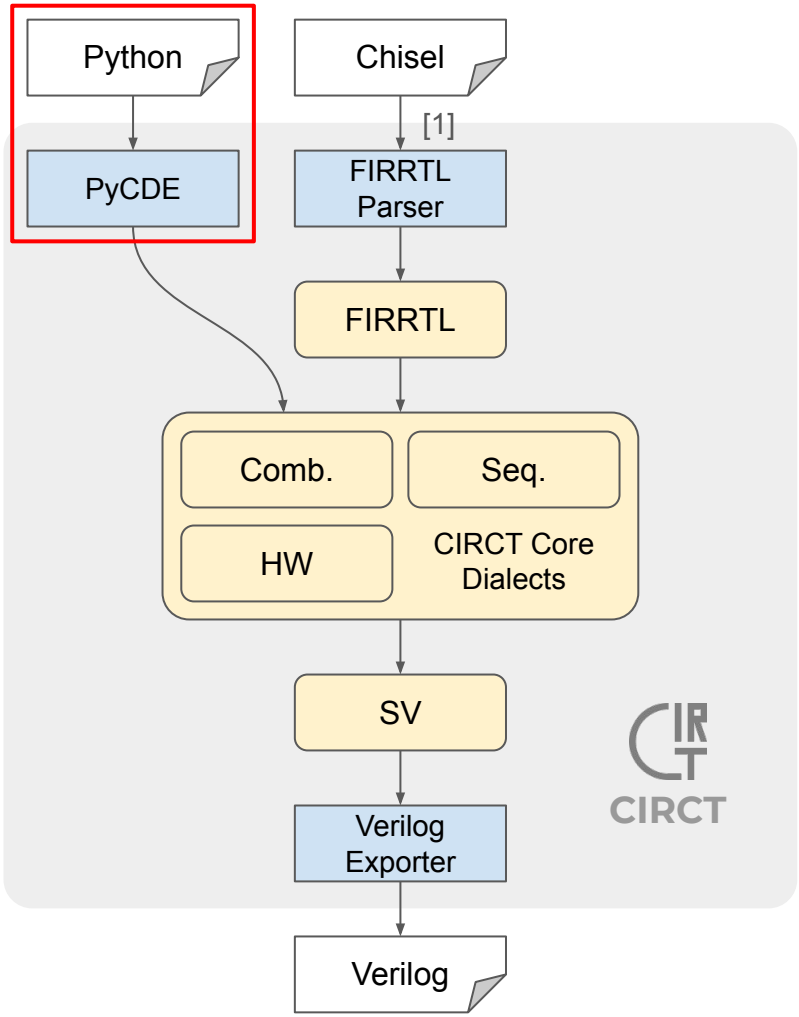
[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

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

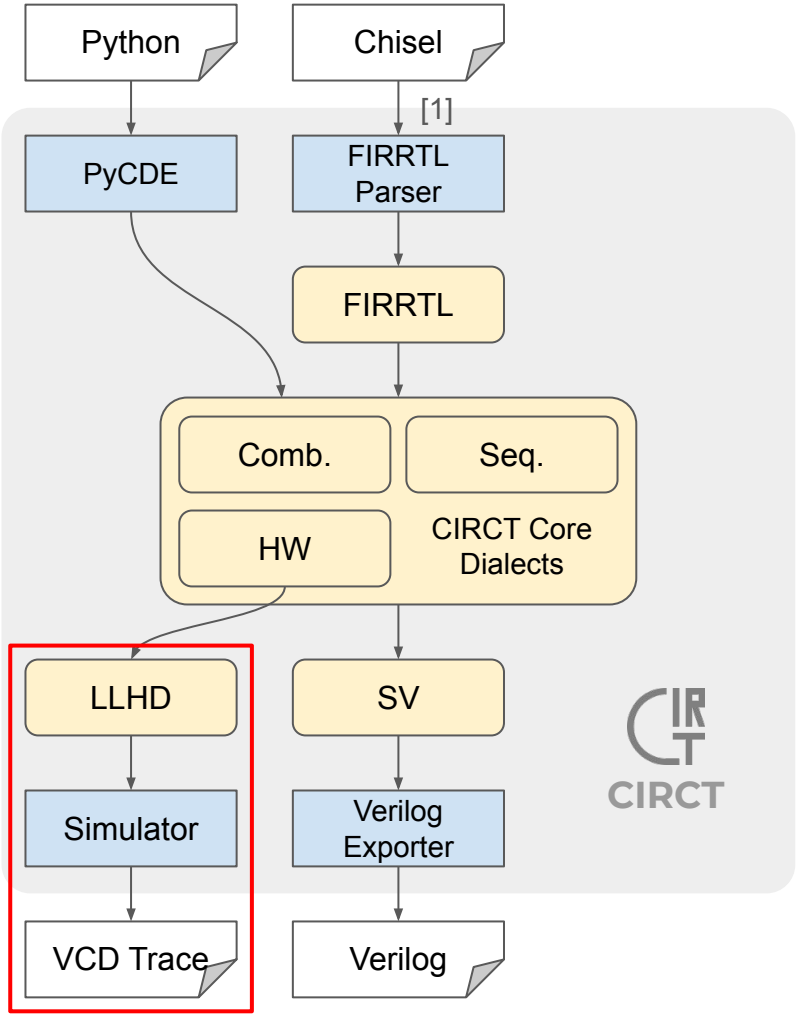
LLHD (Low Level Hardware Description) Dialect

- Dedicated for low-level circuit representation
- Support MLIR-based circuit simulation

⇩ *ongoing*

- Support behavioral, structural, and netlist level simulation for multi-level circuit verification
- Support massive parallelization in the simulation

CIRCT can boost the evolution of hardware simulation techniques

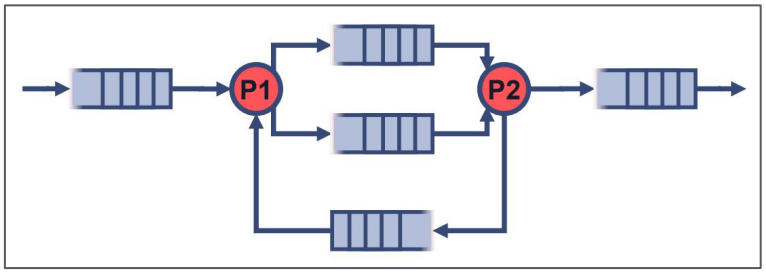


[1] Chisel3: https://github.com/chipsalliance/chisel3
[2] Polygeist: https://github.com/wsmoses/Polygeist

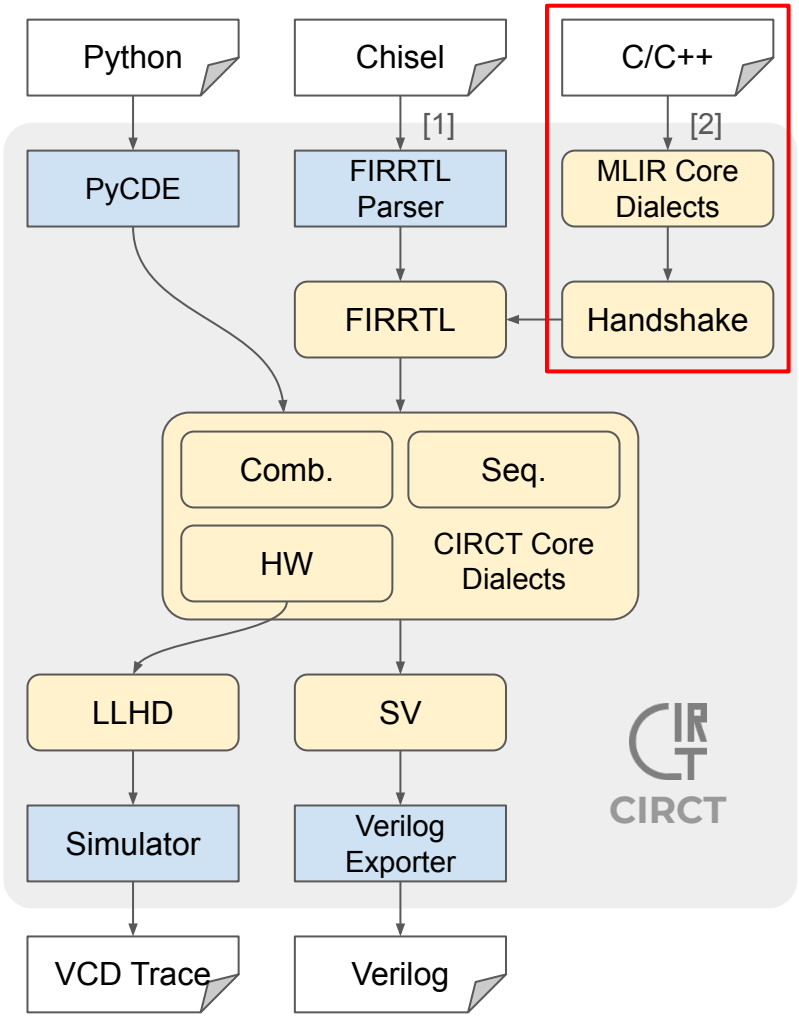
Modular & Extensible: HLS

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

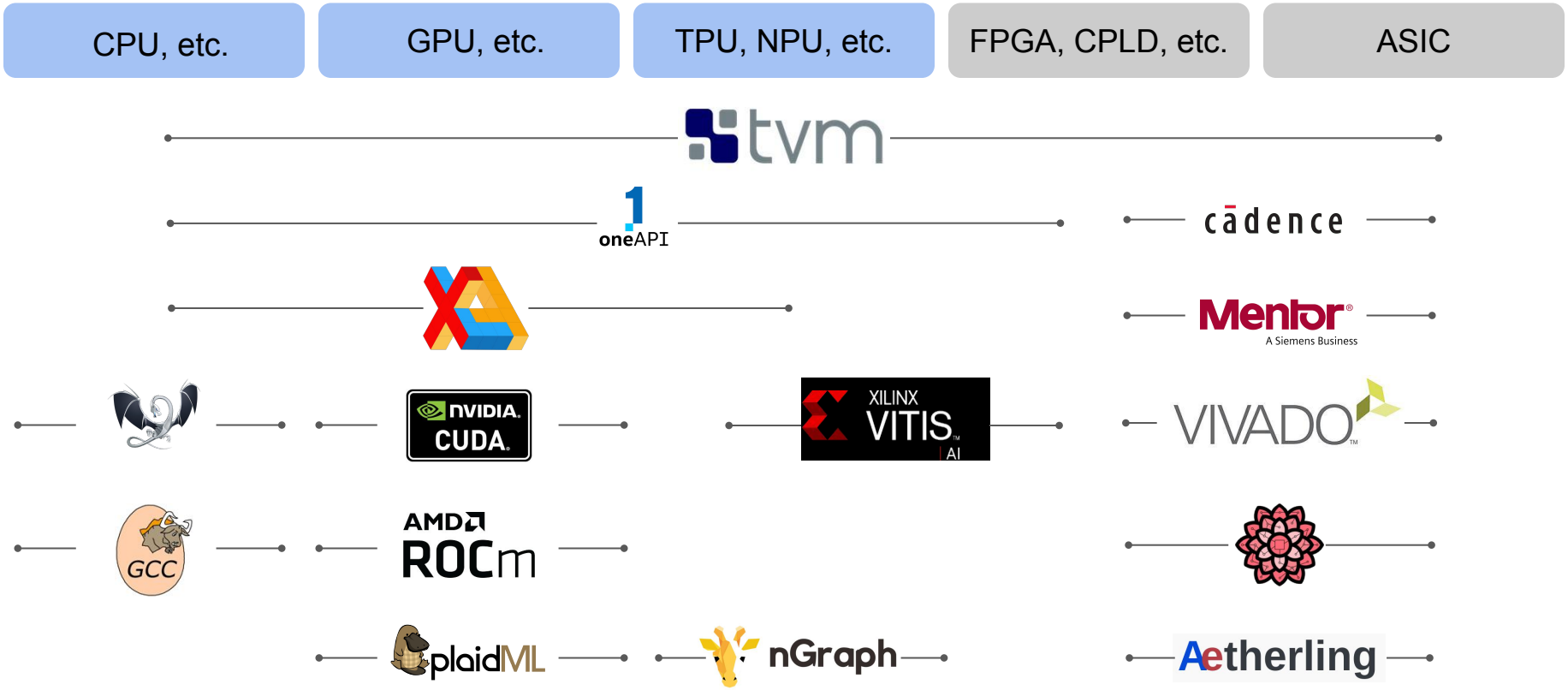
Join the CIRCT Community!

- 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

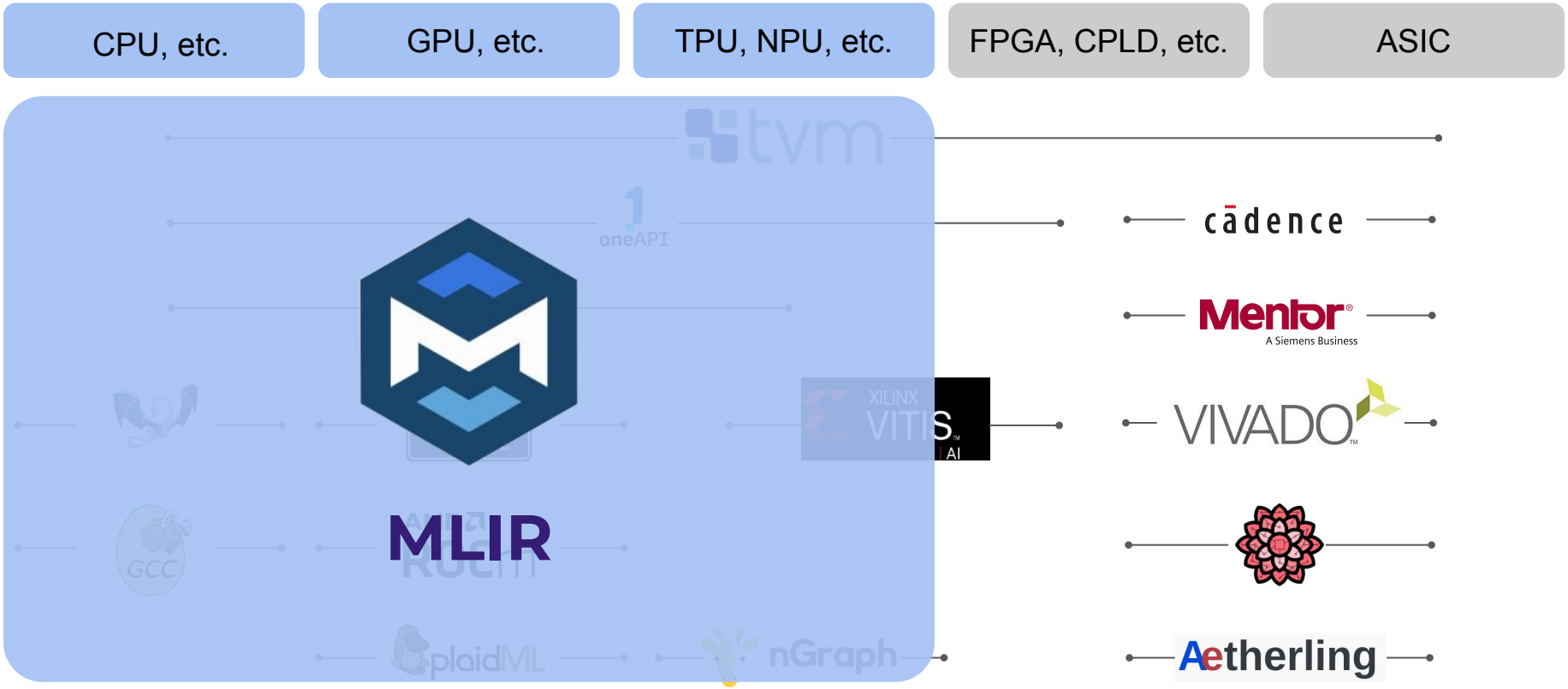
- **Background:** Domain-Specific Architecture (DSA)
- **Software Compilation:** How to program DSA?
  - MLIR: Multi-Level Intermediate Representation
- **Hardware Compilation:** How to design and verify DSA?
  - CIRCT: Circuit IR Compilers and Tools
- **Conclusion:** Software and Hardware Co-design

Spectrum of Compilers



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

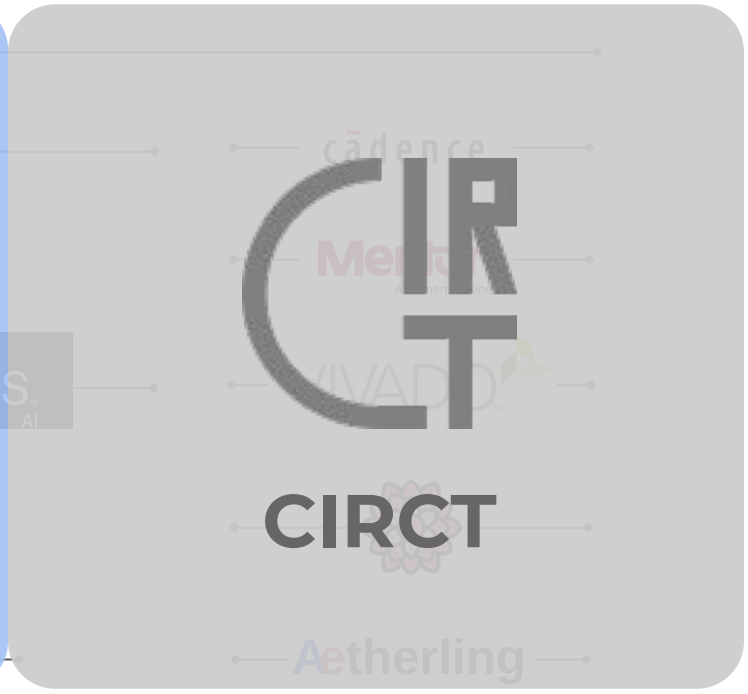
Spectrum of Compilers (Cont'd)



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

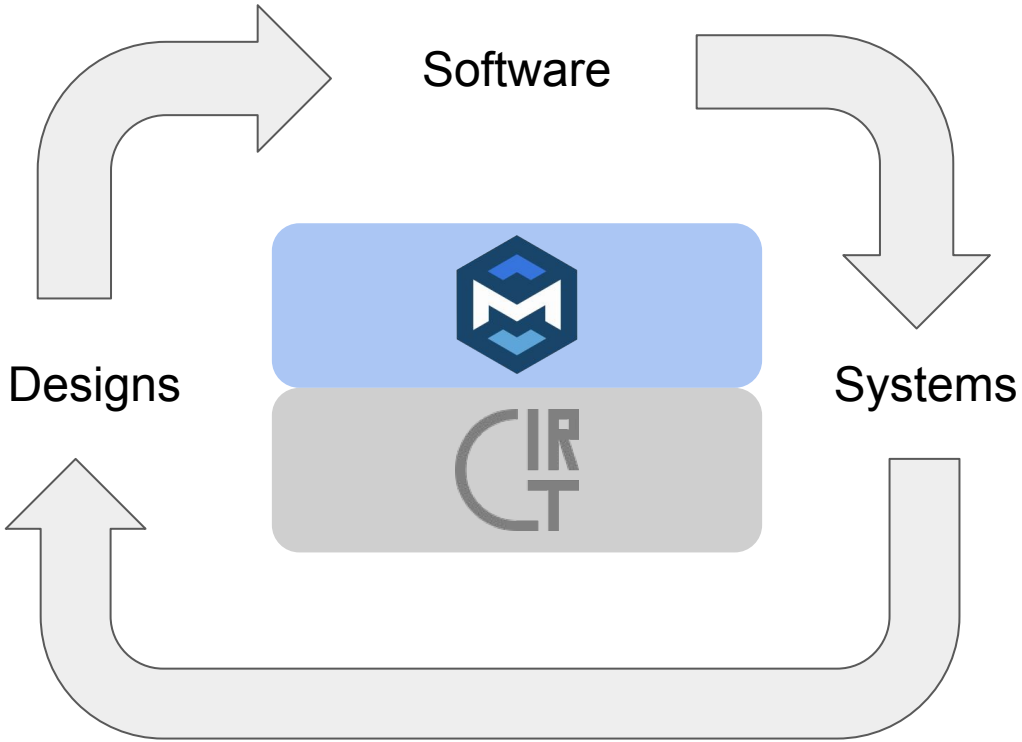
Spectrum of Compilers (Cont'd)

CPU, etc.	GPU, etc.	TPU, NPU, etc.	FPGA, CPLD, etc.	ASIC



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
Dec. 2, 2021