Applying Circuit IR Compilers and Tools (CIRCT) to ML Applications

MLSys Chips and Compilers Symposium

4/5/2021

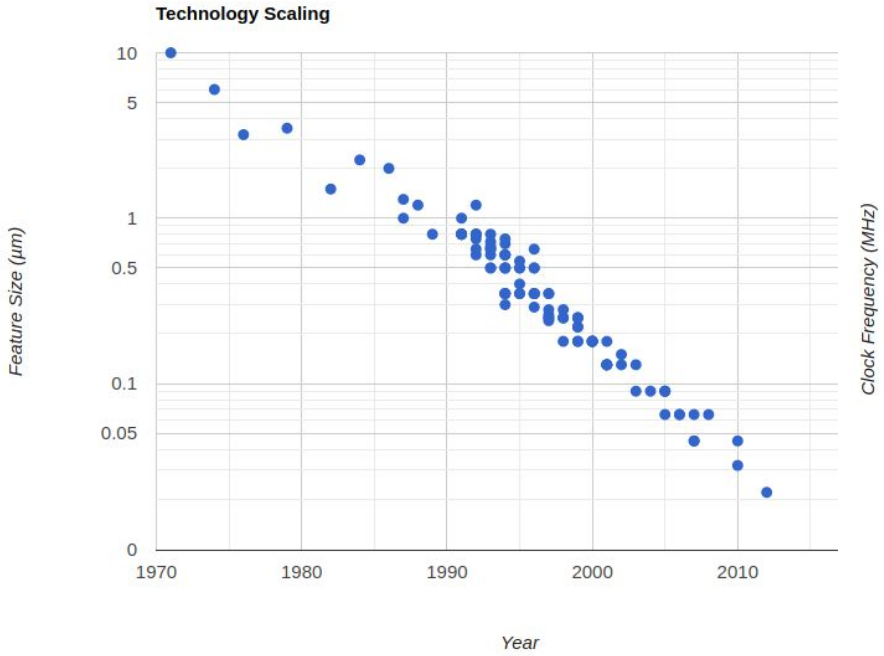
mike@alloystack.io

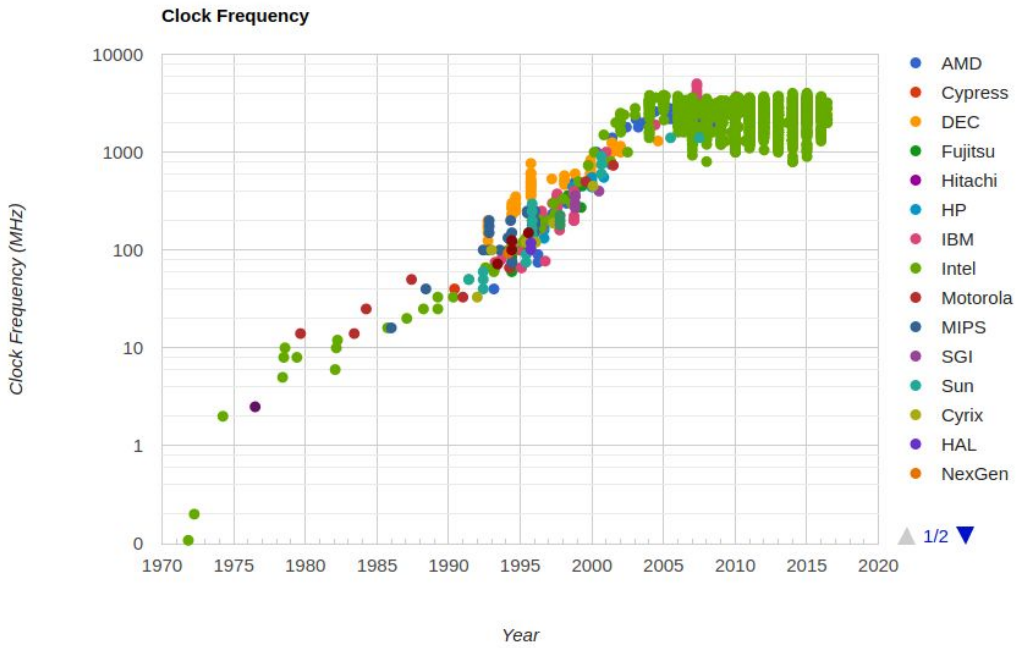
Overview

- Part 1: The End of Moore’s Law
- Part 2: Compilers for the End of Moore’s Law
- Part 3: CIRCT challenges and opportunities

Part 1: The End of Moore’s Law

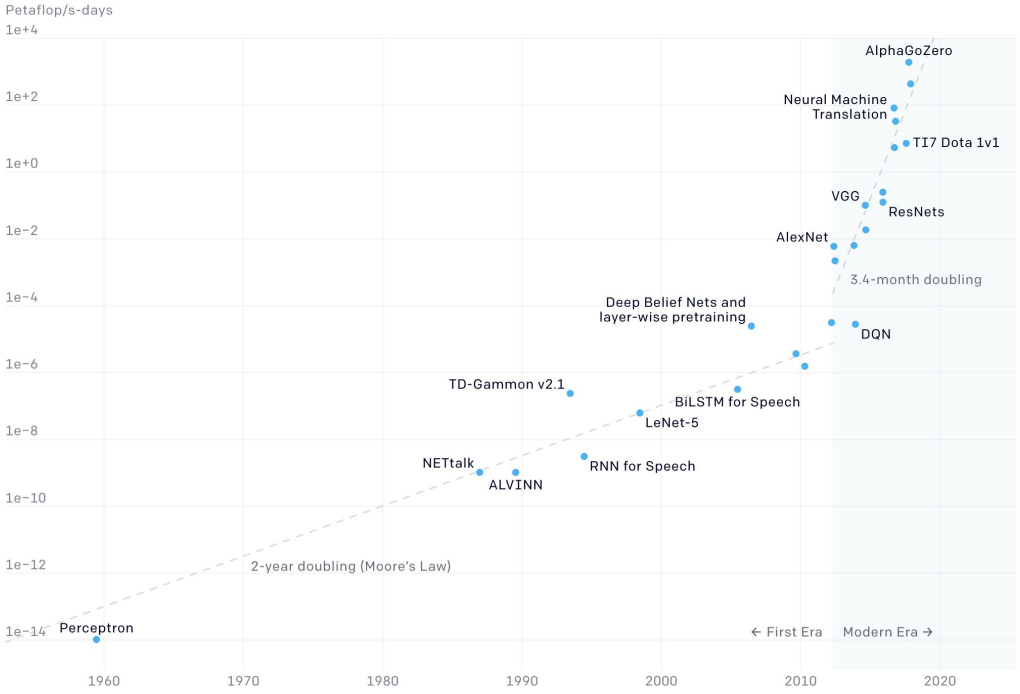
Moore's Law and End of Dennard Scaling





http://cpudb.stanford.edu/visualize

Growing Computational Demands of Machine Learning

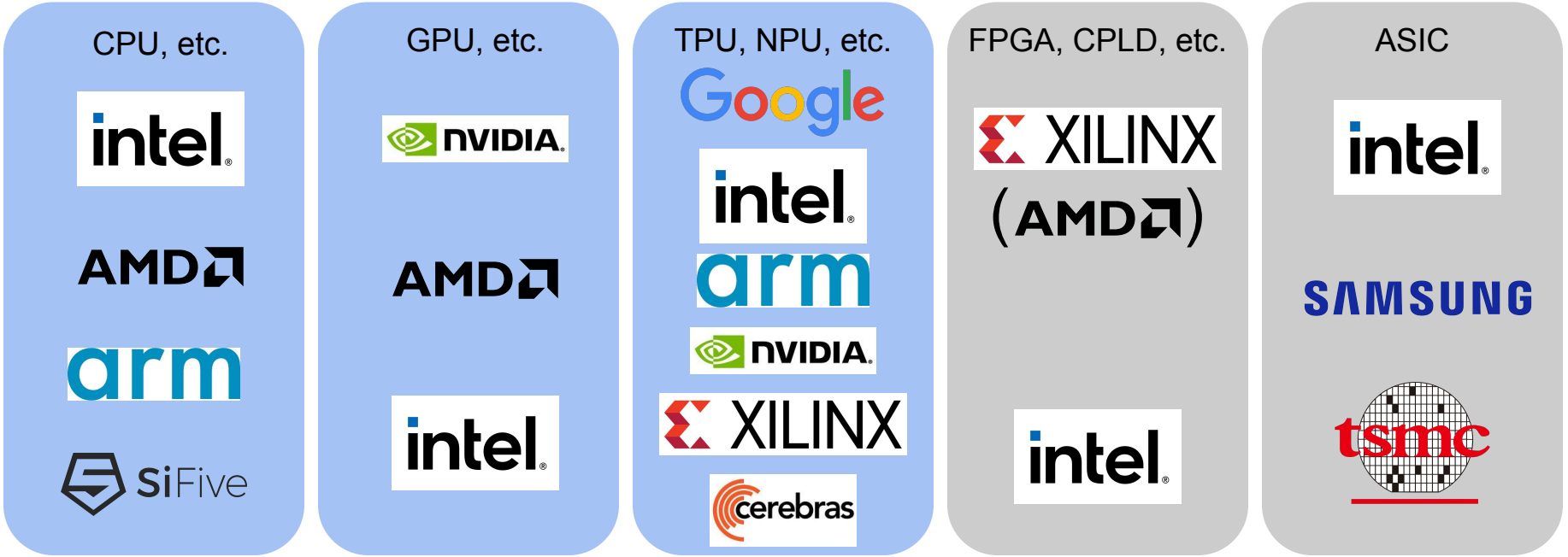


https://openai.com/blog/ai-and-compute

Specialized Hardware for Machine Learning

Fixed processor

Custom processor



Specialization

https://basicmi.github.io/AI-Chip/

The Way Forward

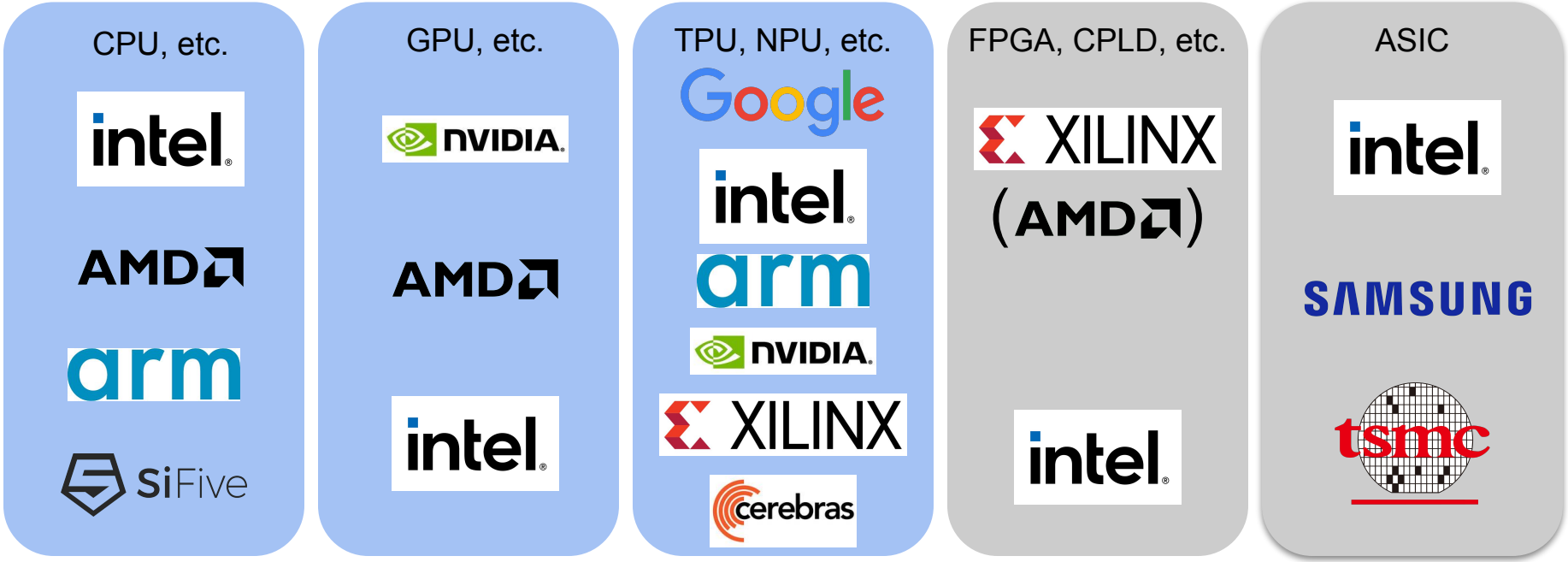
- Economic realities driving the need for specialized hardware
- Designing and programming accelerators is challenging
- Specialized hardware needs specialized compilers
- Specialized compilers need good compiler infrastructure

Part 2: Compilers for the End of Moore’s Law

Prior Art

Fixed processor

Custom processor



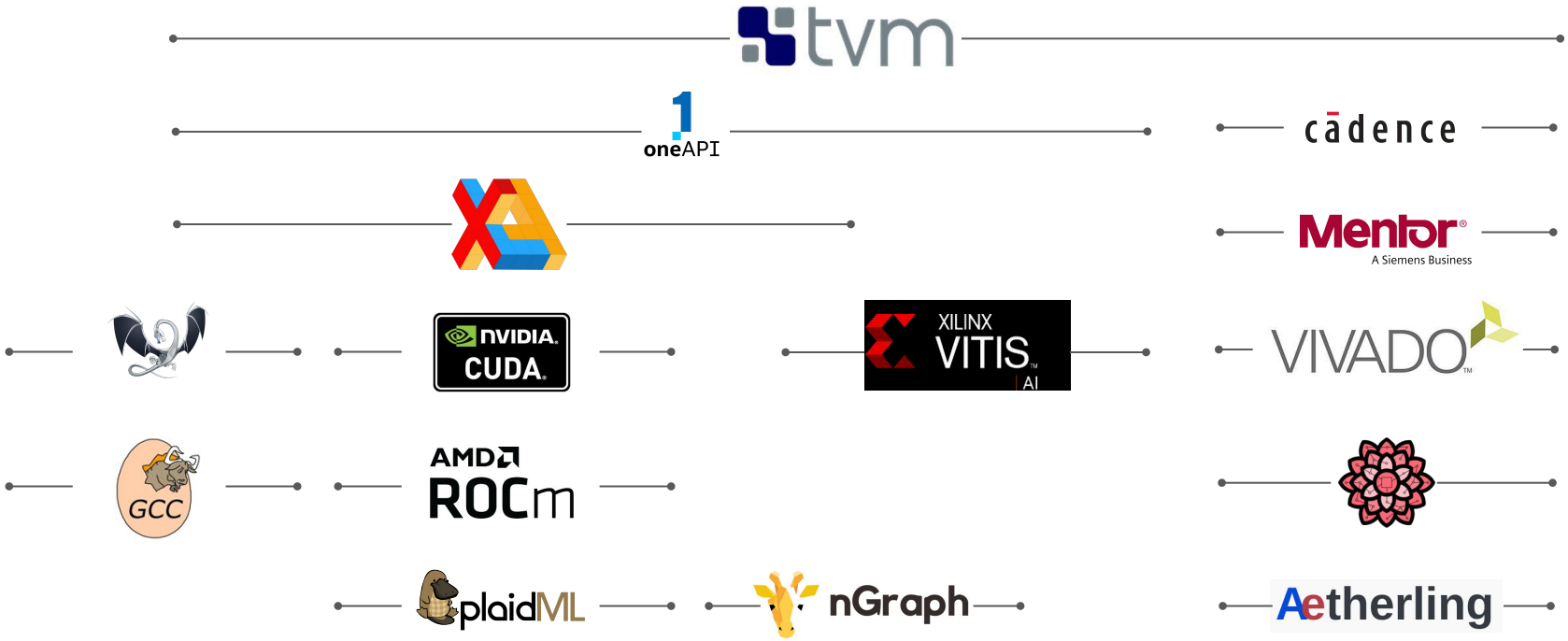
Specialization

Prior Art

Fixed processor

Custom processor

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



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, library based philosophy
- Modular and extensible
- Originally created within Google for compiling TensorFlow
- Sufficiently general to compile lots of domains besides ML

https://mlir.llvm.org

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

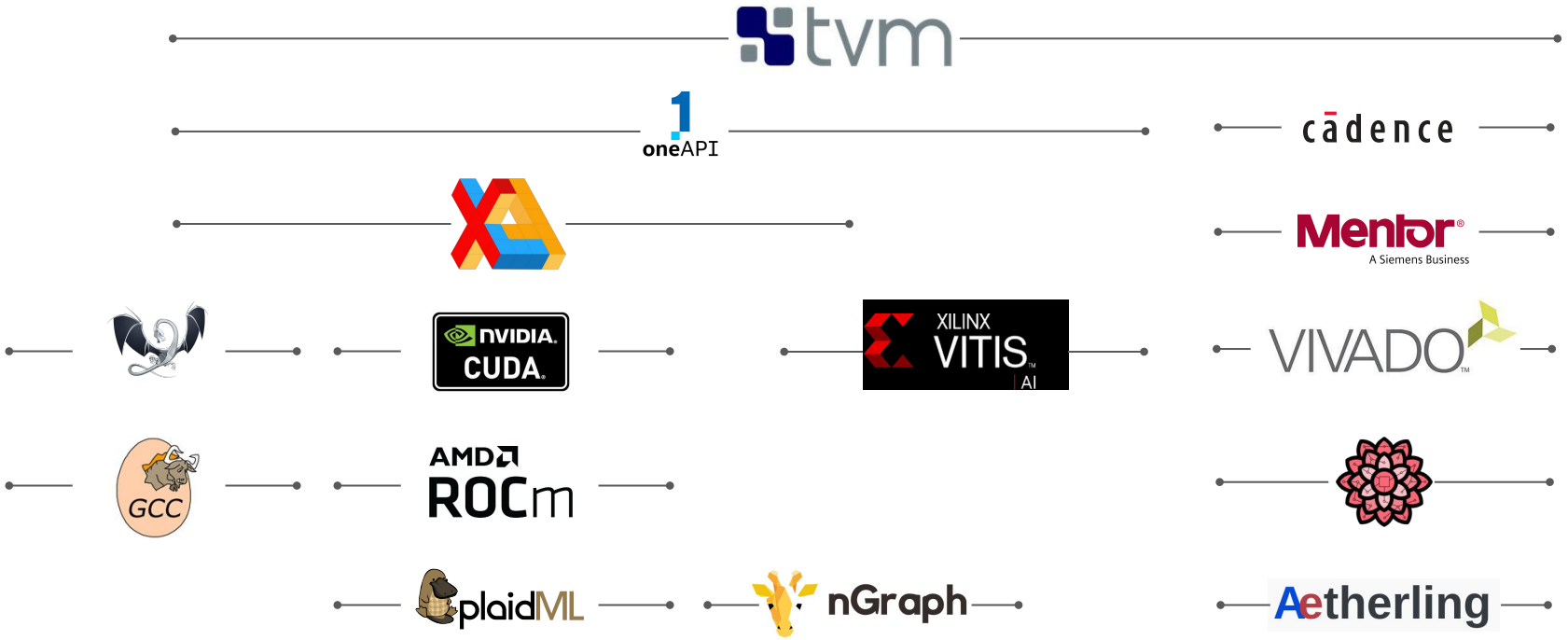
https://circt.llvm.org

How MLIR and CIRCT Fit In

Fixed processor

Custom processor

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



How MLIR and CIRCT Fit In

Fixed processor

Custom processor

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

cadence

Mentor A Siemens Business

VIVADO

Aetherling

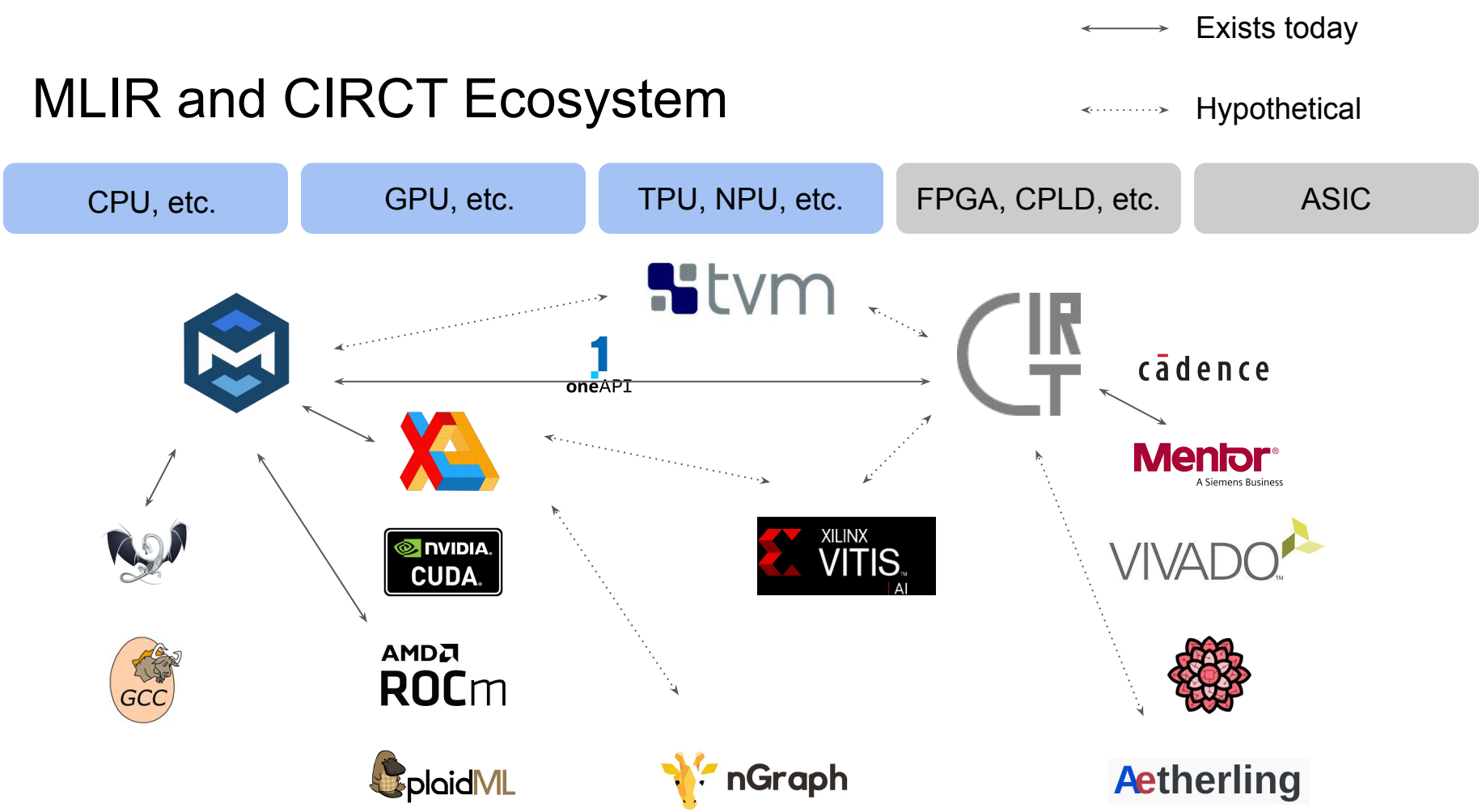
How MLIR and CIRCT Fit In

Fixed processor

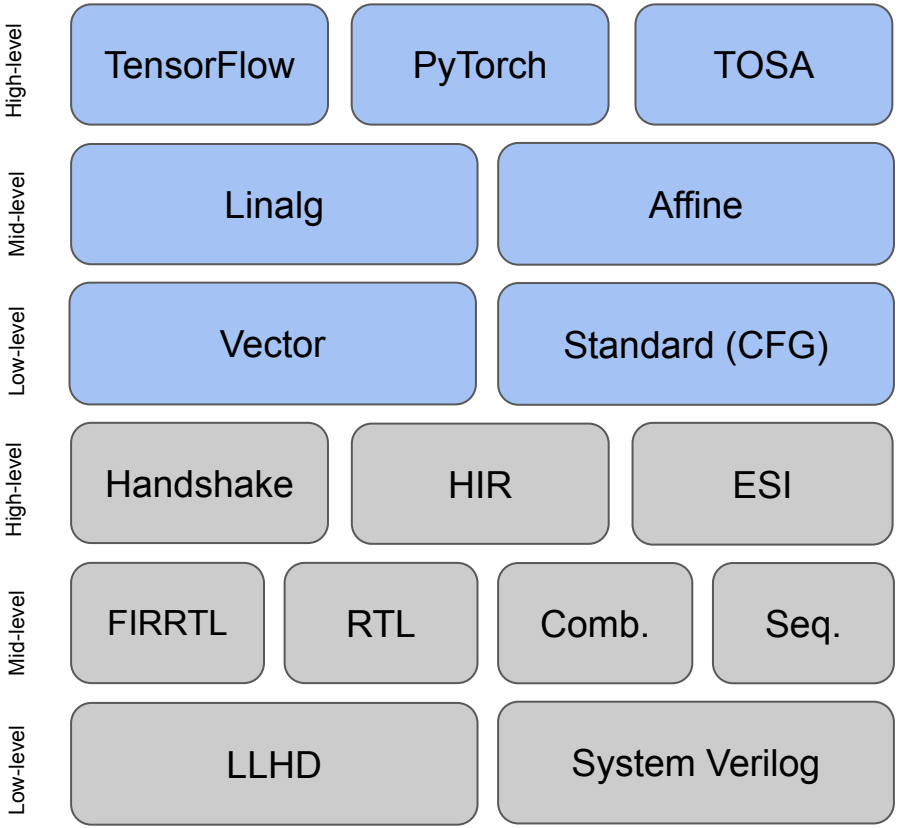
Custom processor

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

MLIR and CIRCT Ecosystem



Compiler Stack

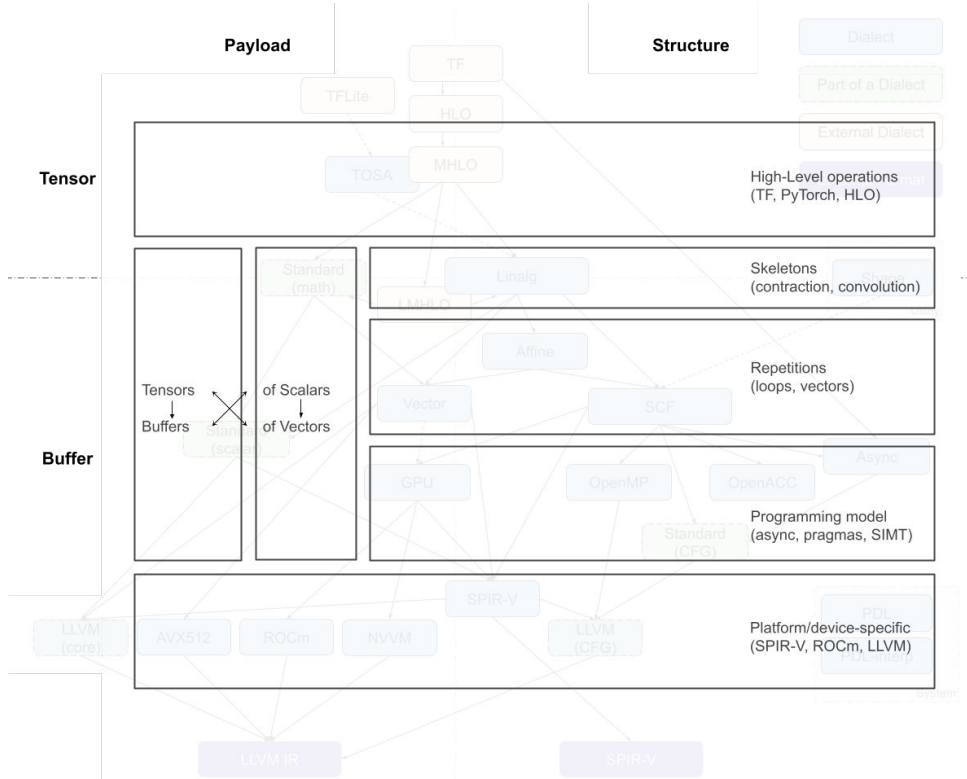


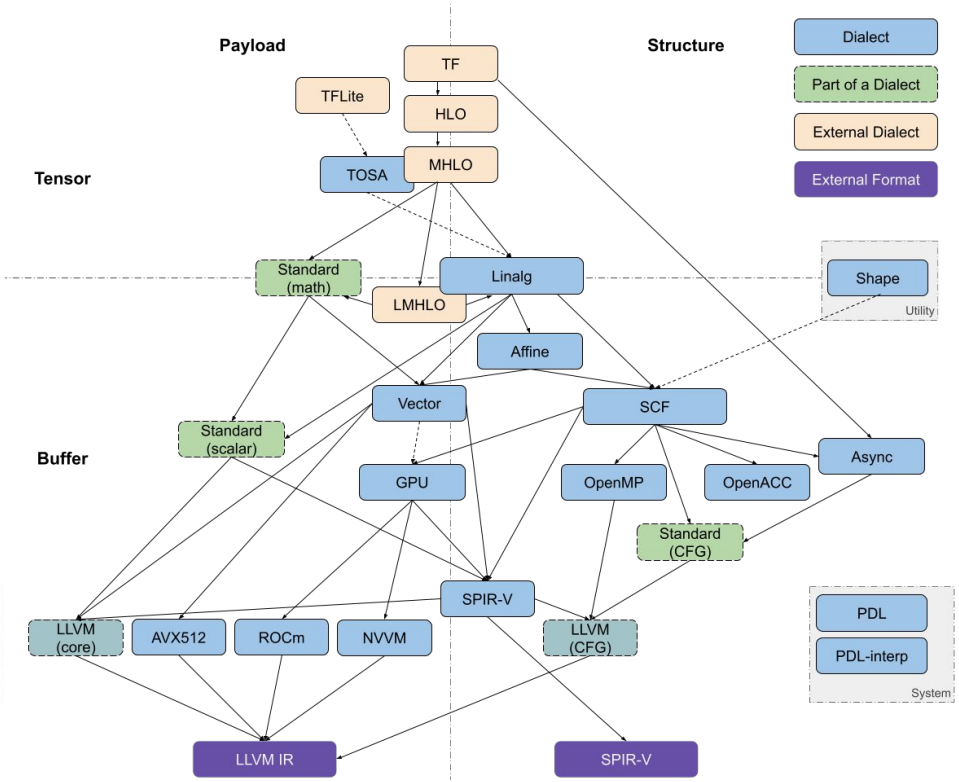
MLIR dialects

CIRCT dialects

- Start from ML frameworks
- Use MLIR for as much as possible
- Get down to common MLIR layers
- Translate those into CIRCT
- Lower to CIRCT core dialects
- Translate those to exit dialects

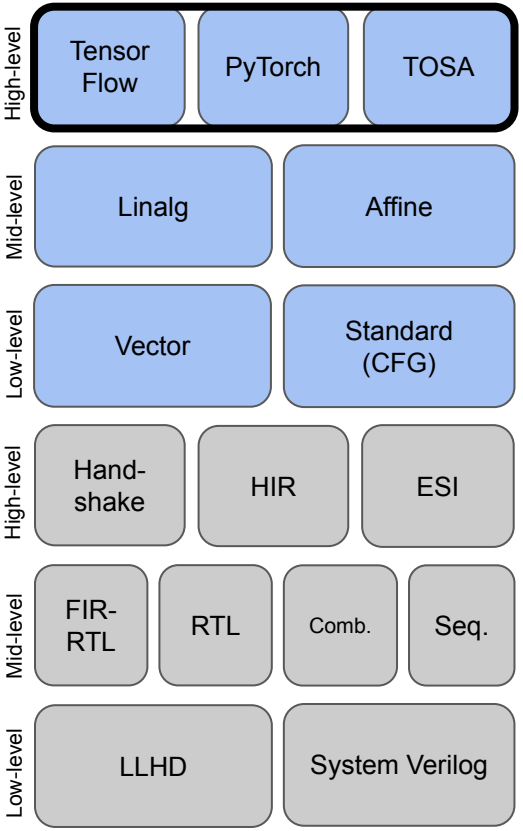
MLIR codegen overview



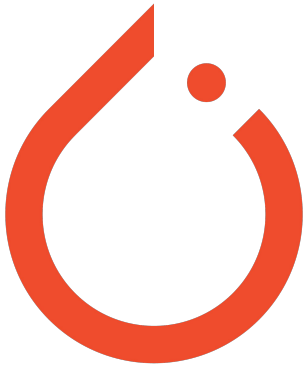


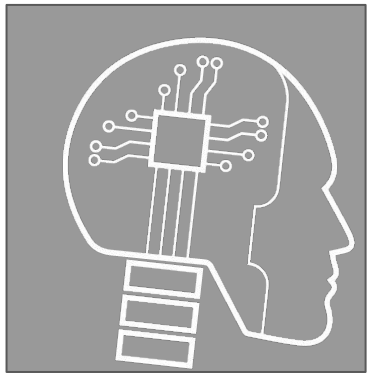
https://llvm.discourse.group/t/codegen-dialect-overview/2723

High-level dialects: TensorFlow, PyTorch, TOSA

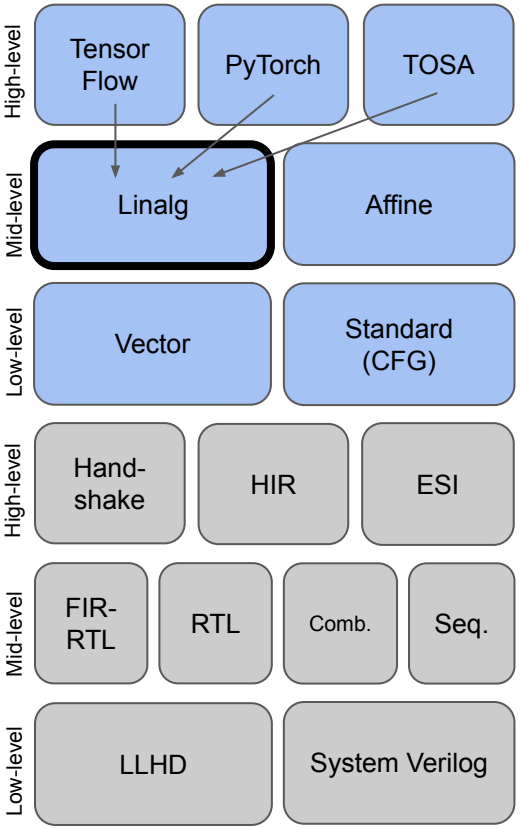



```
func @matmul(%lhs: tensor<5x10xf32>, %rhs: tensor<10x10xf32>) -> (tensor<5x10xf32>) {
  %0 = "tf.BatchMatMulV2"(%lhs, %rhs) : (tensor<5x10xf32>, tensor<10x10xf32>) ->
                                        tensor<5x10xf32>
  return %0 : tensor<5x10xf32>
}
```

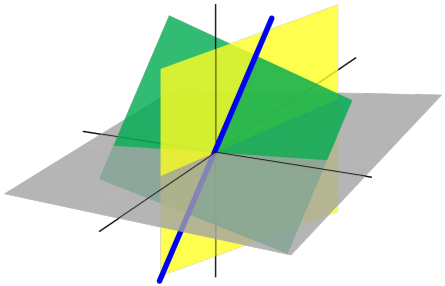




Mid-level dialects: Linalg



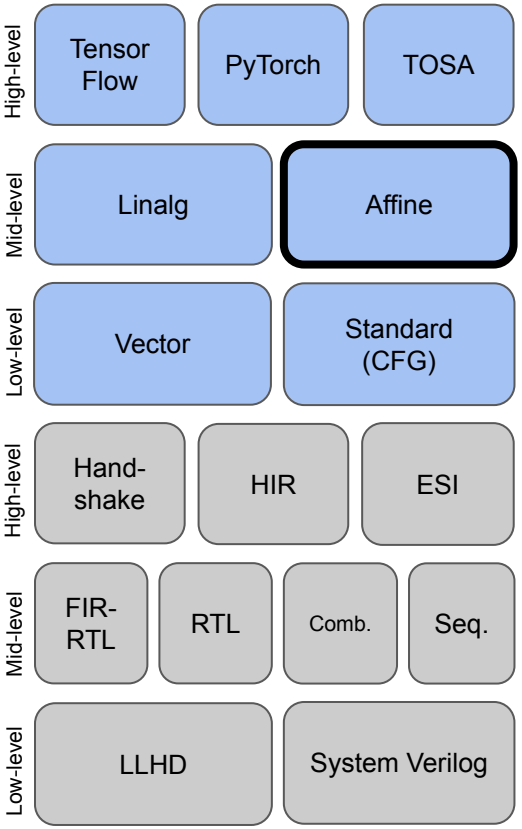

```
func @matmul(%A: tensor<5x10xf32>, %B: tensor<10x10xf32>,
             %C: tensor<10x10xf32>) -> tensor<10x10xf32> {
  %0 = linalg.matmul ins(%A, %B: tensor<5x10xf32>, tensor<10x10xf32>)
                     outs(%C: tensor<10x10xf32>) -> tensor<10x10xf32>
  return %0: tensor<10x10xf32>
}
```



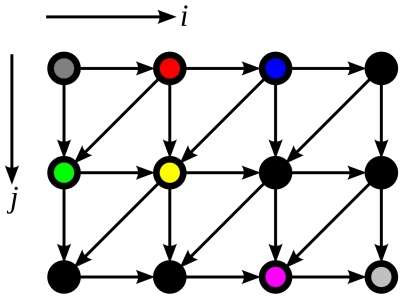
- Builds on lots of prior art
- A handful of generic operations
- Named operations (like above) that map to generic operations

https://mlir.llvm.org/docs/Dialects/Linalg/

Mid-level dialects: Affine



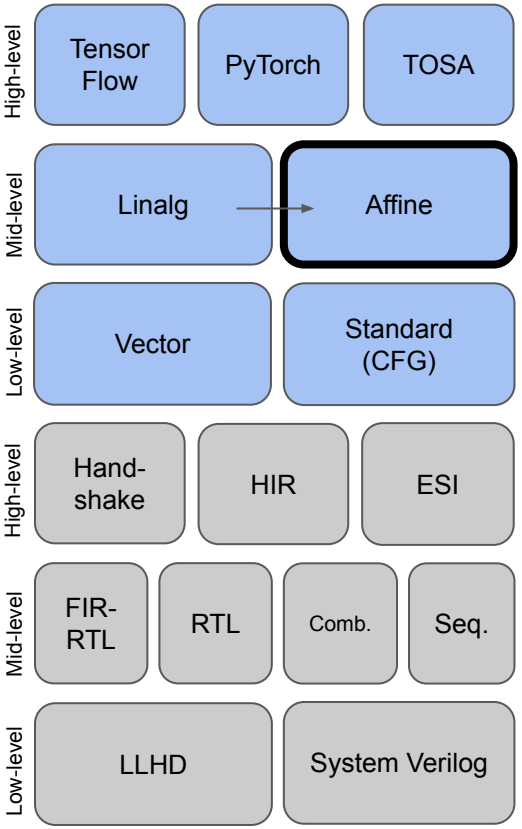

```
func @matmul(%A: memref<5x10xf32>, %B: memref<10x10xf32>,
             %C: memref<10x10xf32>) -> memref<10x10xf32> {
  affine.for %arg3 = 0 to 5 {
    affine.for %arg4 = 0 to 10 {
      affine.for %arg5 = 0 to 10 { ... }
    }
  }
  return %0 : memref<10x10xf32>
}
```



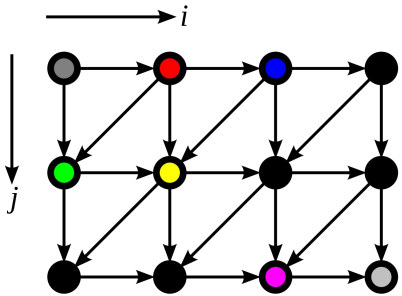
- Polyhedral compilation toolbox
- Affine expressions, maps, and integer sets
- Affine loops and transformations

https://mlir.llvm.org/docs/Dialects/Affine/

Mid-level dialects: Affine



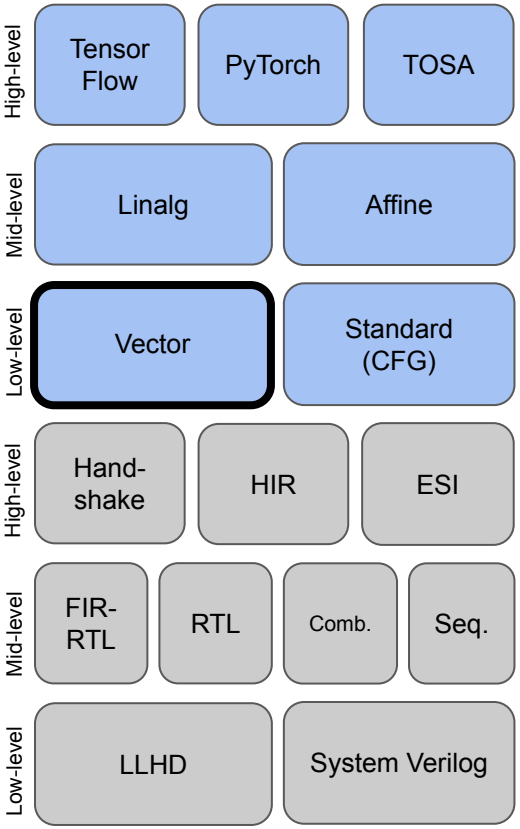

```
func @matmul(%A: memref<5x10xf32>, %B: memref<10x10xf32>,
             %C: memref<10x10xf32>) -> memref<10x10xf32> {
  affine.for %arg3 = 0 to 5 {
    affine.for %arg4 = 0 to 10 {
      affine.for %arg5 = 0 to 10 { ... }
    }
  }
  return %0 : memref<10x10xf32>
}
```



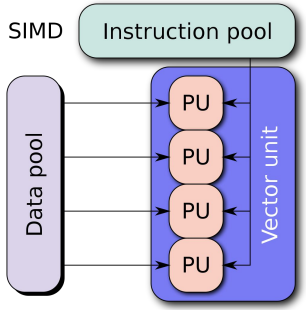
- Polyhedral compilation toolbox
- Affine expressions, maps, and integer sets
- Affine loops and transformations

https://mlir.llvm.org/docs/Dialects/Affine/

Low-level dialects: Vector



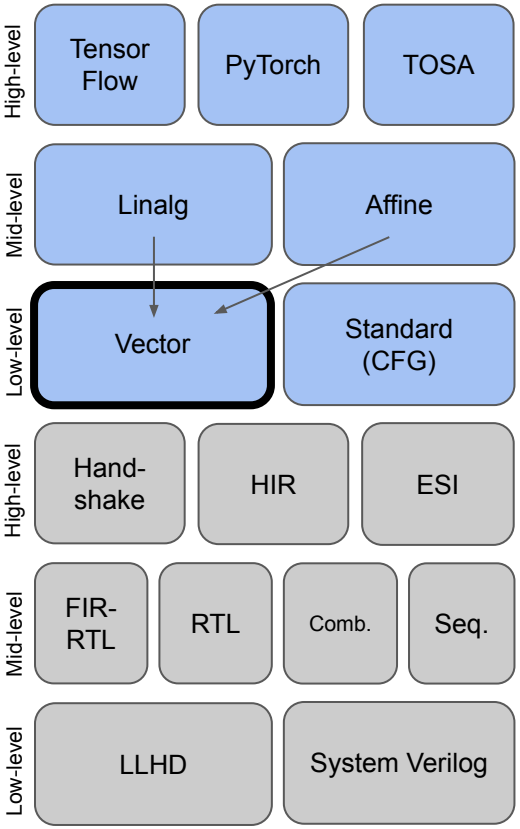

```
func @transpose(%arg0: vector<3x7xf32>) -> vector<7x3xf32> {
  %0 = vector.transpose %arg0, [1, 0] : vector<3x7xf32> to vector<7x3xf32>
  return %0 : vector<7x3xf32>
}
```



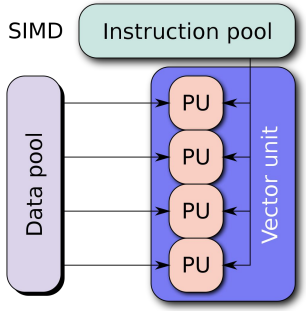
- Virtual n-D vectors and transformations on them
- Hardware vectors that match a specific processor's ISA

https://mlir.llvm.org/docs/Dialects/Vector/

Low-level dialects: Vector



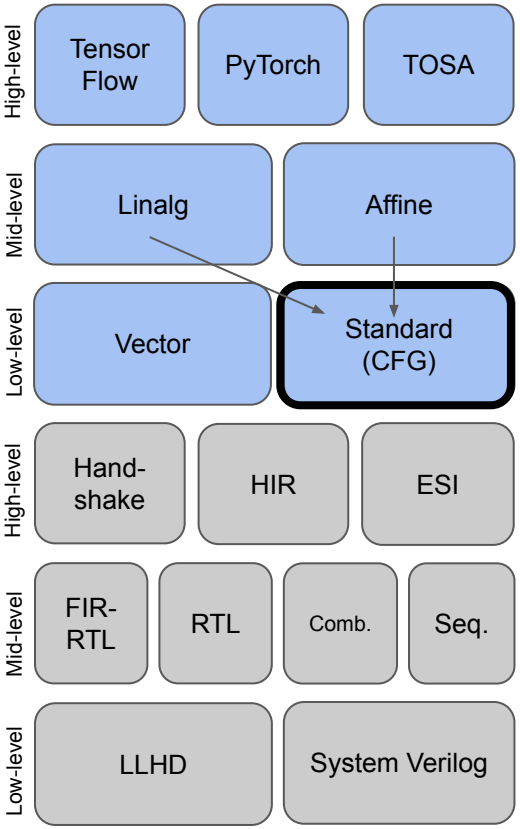

```
func @transpose(%arg0: vector<3x7xf32>) -> vector<7x3xf32> {
  %0 = vector.transpose %arg0, [1, 0] : vector<3x7xf32> to vector<7x3xf32>
  return %0 : vector<7x3xf32>
}
```



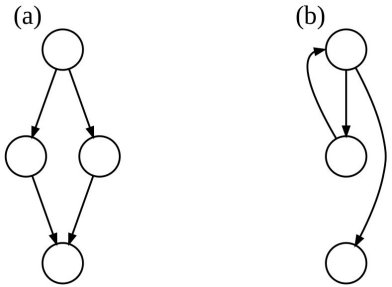
- Virtual n-D vectors and transformations on them
- Hardware vectors that match a specific processor's ISA

https://mlir.llvm.org/docs/Dialects/Vector/

Low-level dialects: Standard control-flow graph



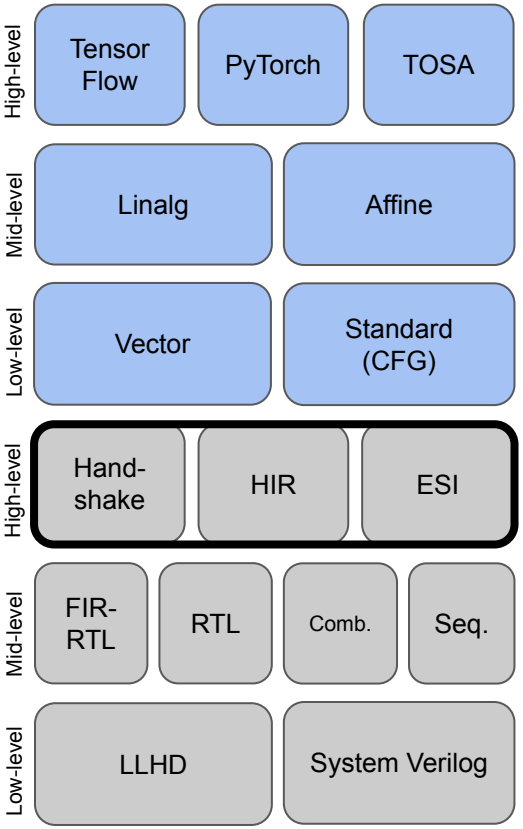

```
func @simple_loop() {
^bb0:
  br ^bb1
^bb1:   // pred: ^bb0
  %c1 = constant 1 : index
  %c42 = constant 42 : index
  br ^bb2(%c1 : index)
^bb2(%0: index):        // 2 preds: ^bb1, ^bb3
  %1 = cmpi slt, %0, %c42 : index
  cond_br %1, ^bb3, ^bb4
```

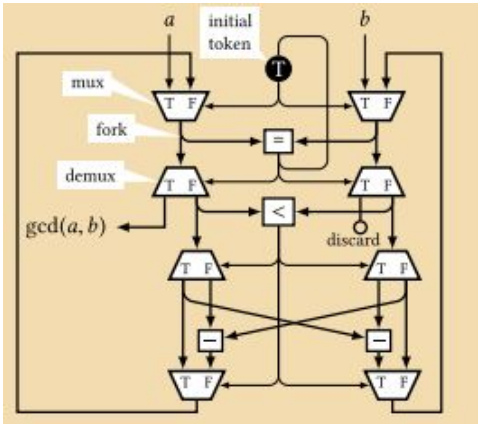


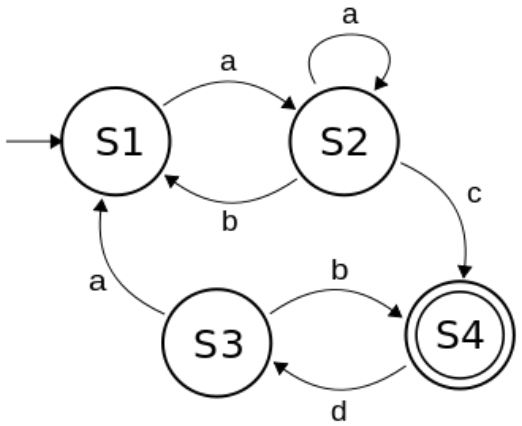
- MLIR's standard dialect
- Basic blocks and branches
- Evolution of LLVM IR

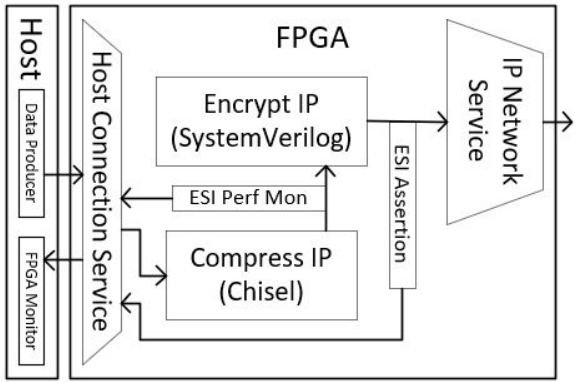
https://mlir.llvm.org/docs/Dialects/Standard/

High-level dialects

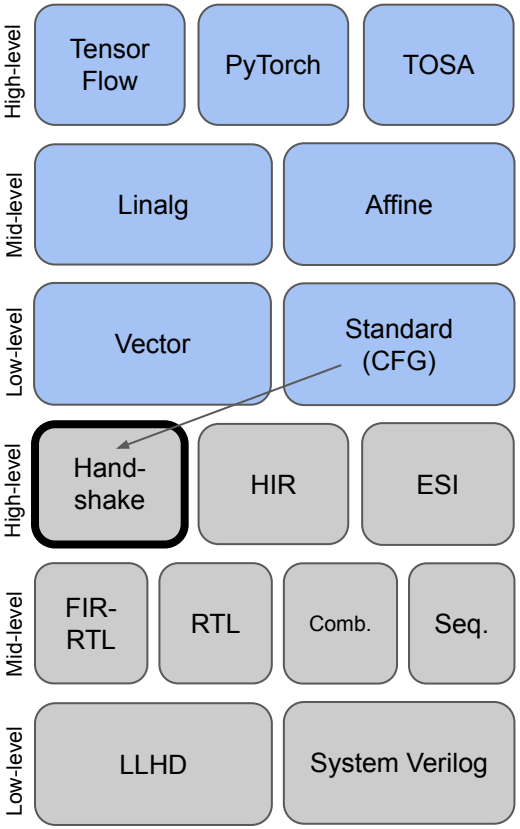




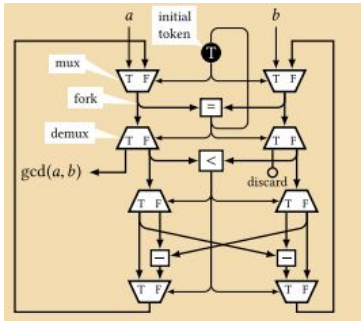




High-level dialects: Handshake



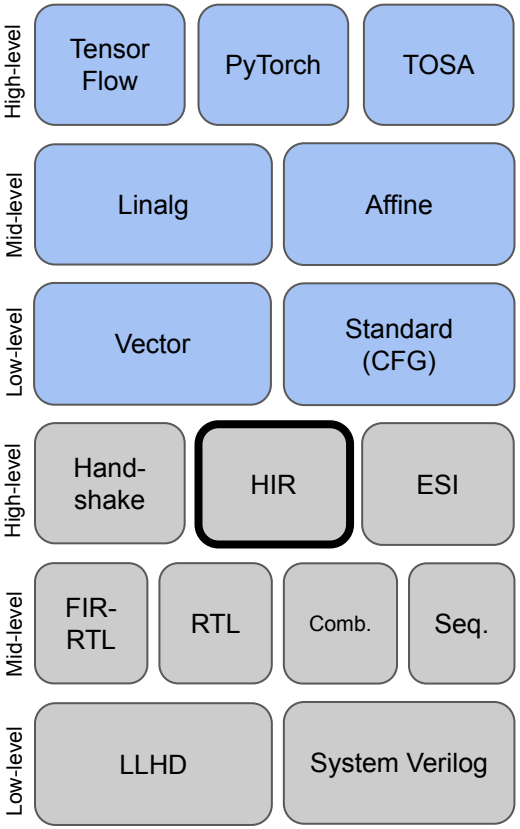

```
handshake.func @fork(%arg0: none, %arg1: none) -> (none, none, none) {
  %0:2 = "handshake.fork"(%arg0) {control = true} : (none) -> (none, none)
  %1 = "handshake.join"(%0#0, %0#1) {control = true}: (none, none) -> none
  handshake.return %1, %arg1 : none, none, none
}
```



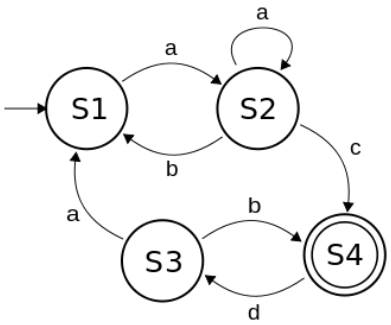
- Composable set of control flow operators like fork and join
- Rooted in Kahn Process Networks and Petri Nets
- Efficient hardware implementation

https://circt.llvm.org/docs/Dialects/Handshake/

High-level dialects: HIR



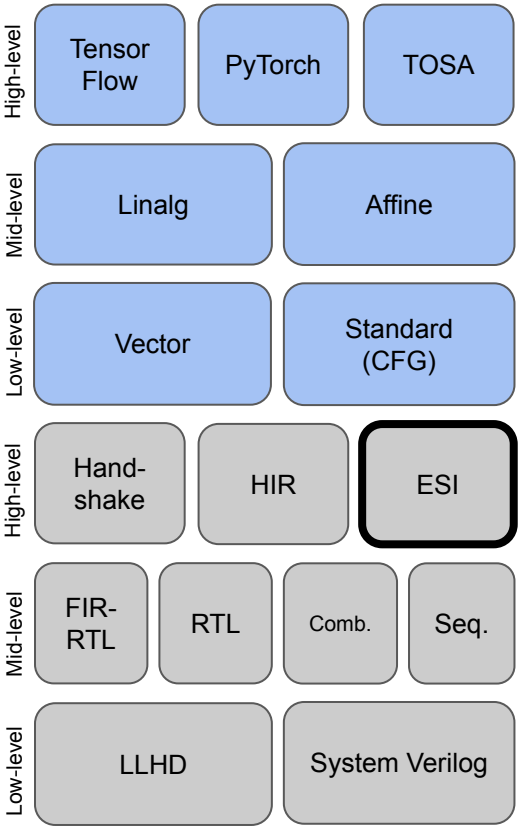

```
hir.func @mac at %t (%a :i32, %b :i32, %c :i32) -> (i32 delay 3) {
  %1 = hir.constant 1
  %2 = hir.constant 2
  %m = hir.call @mult_3stage (%a,%b) at %t : (i32, i32) -> (i32 delay 3)
  %c2= hir.delay %c by %2 at %t : i32 -> i32
  %res = hir.add (%m,%c2) : (i32, i32) -> (i32)
  %res1 = hir.delay %res by %1 at %t offset %2 : i32 -> i32
  hir.return (%res1) : (i32)
}
```



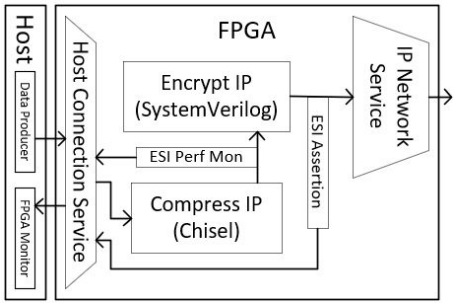
- Zero cost abstractions
- Represents finite state machines in a high-level way
- Supports banked memories

[HIR: An MLIR-based IR for Hardware Accelerator Description]

High-level dialects: Elastic Silicon Interconnect (ESI)



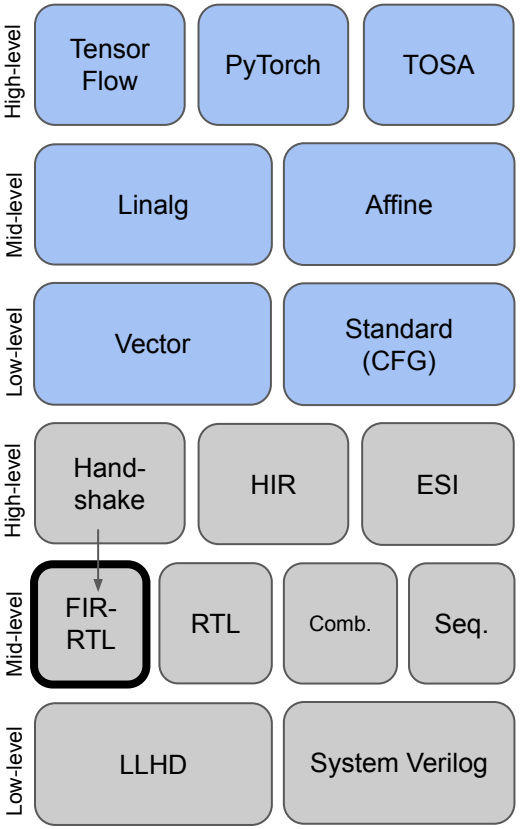

```
rtl.module @esi(%clock: i1, %reset: i1) {
  %esiChannel, %0 = rtl.instance "sender" @Sender (%clock) : (i1) -> (!esi.channel<i4>, i8)
  %bufferedChannel = esi.buffer %clock, %reset, %esiChannel { stages = 4 } : i4
  rtl.instance "receiver" @Reciever (%bufferedChannel, %clock) : (!esi.channel<i4>, i1) -> ()
}
```



- Type system for channels in hardware
- Connect on-chip and off-chip components
- Generates elastic connector circuits
- Supports cosimulation

https://circt.llvm.org/docs/Dialects/ESI/

Mid-level dialects: FIRRTL

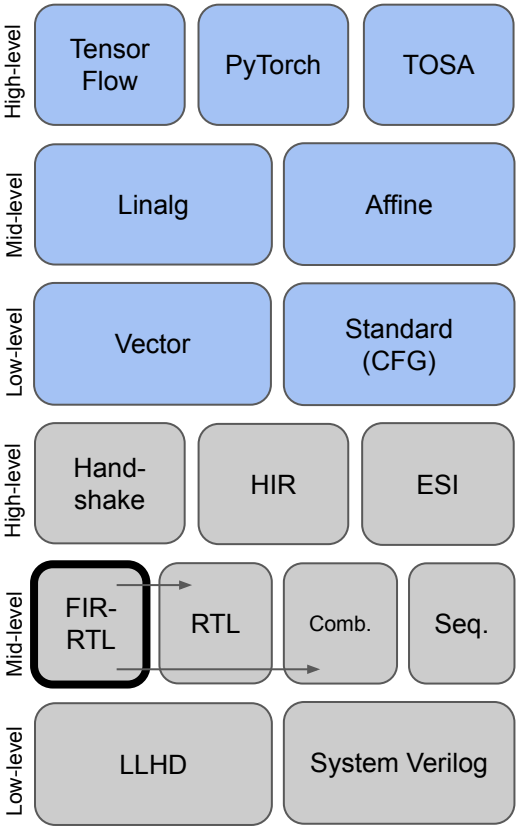



```
firrtl.module @bundle0(%a : !firrtl.bundle<f1: uint<1>, f2: flip<sint<1>>>,
                       %b : !firrtl.bundle<f1: flip<uint<1>>, f2: sint<1>>) {
  firrtl.connect %b, %a : !firrtl.bundle<f1: flip<uint<1>>, f2: sint<1>>,
                          !firrtl.bundle<f1: uint<1>, f2: flip<sint<1>>>
}
```

- Core IR from Chisel compiler
- Abstractions that are below Chisel but above System Verilog
- Lots of effort to improve on the design in CIRCT's core dialects

https://circt.llvm.org/docs/Dialects/FIRRTL/

Mid-level dialects: FIRRTL

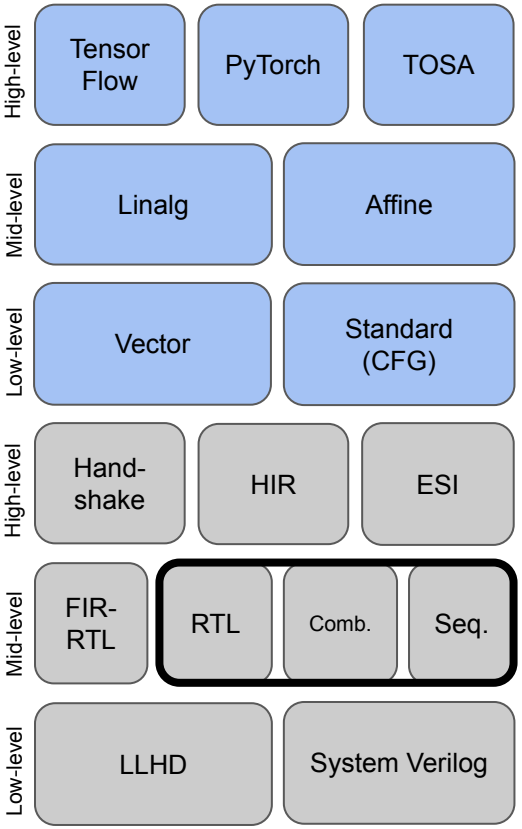



```
firrtl.module @bundle0(%a : !firrtl.bundle<f1: uint<1>, f2: flip<sint<1>>>,
                       %b : !firrtl.bundle<f1: flip<uint<1>>, f2: sint<1>>) {
  firrtl.connect %b, %a : !firrtl.bundle<f1: flip<uint<1>>, f2: sint<1>>,
                          !firrtl.bundle<f1: uint<1>, f2: flip<sint<1>>>
}
```

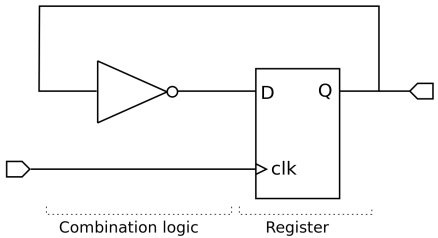
- Core IR from Chisel compiler
- Abstractions that are below Chisel but above System Verilog
- Lots of effort to improve on the design in CIRCT's core dialects

https://circt.llvm.org/docs/Dialects/FIRRTL/

Mid-level dialects: RTL, Combinational, Sequential



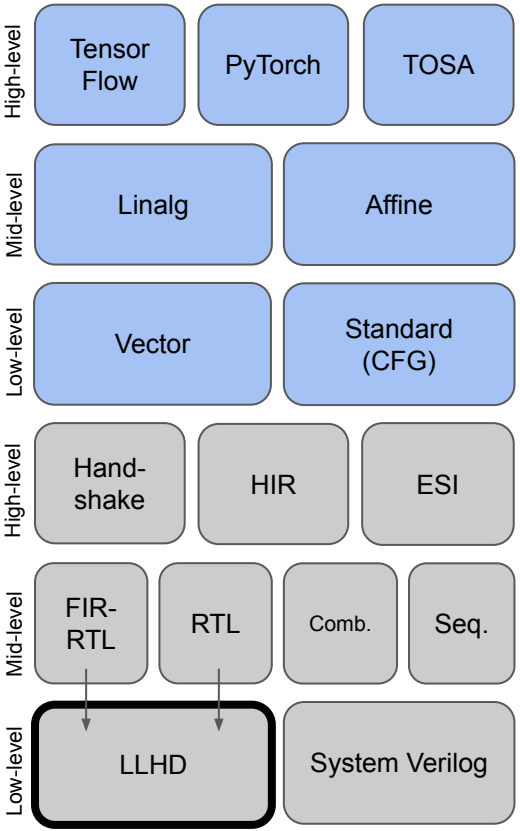

```
rtl.module @test1(%a: i12) -> (i32){
  %b = comb.add %a, %a : i12
  %c = comb.mul %a, %b : i12
  %d = comb.sext %c : (i12) -> i32
  rtl.output %d : i32
}
```



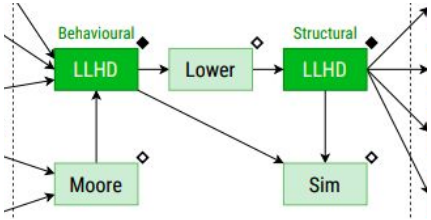
- Structure (modules and ports)
- Combinational logic (and, or, xor)
- Sequential logic (registers)
- Work in progress

https://circt.llvm.org/docs/Dialects/RTL/

Low-level dialects: LLHD



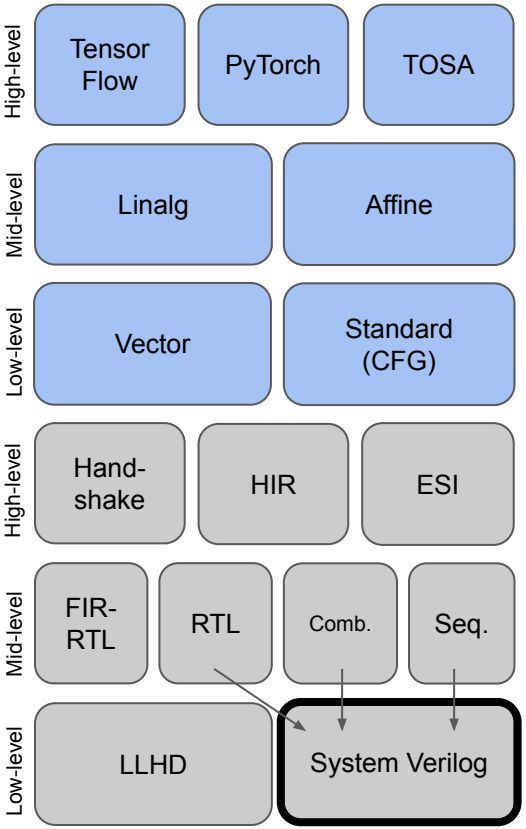

```
llhd.entity @top (%arg0 : !llhd.sig<i1>) -> (%arg1 : !llhd.sig<i1>) {
  %t = #llhd.time<1ns, 2d, 3e> : !llhd.time
  %0 = llhd.prb %arg0 : !llhd.sig<i1>
  llhd.drv %arg1, %0 after %t : !llhd.sig<i1>
}
```



- Low-level, even for hardware
- Supports 9-valued logic
- Encodes time in the type system
- Includes MLIR-based simulator

https://circt.llvm.org/docs/Dialects/LLHD/

Low-level dialects: System Verilog




```
rtl.module @sytemverilog(%clock: i1, %reset: i1) {
  %c0 = rtl.constant 0 : i32
  %c42 = rtl.constant 42 : i32
  %reg = sv.reg : !rtl.inout<i32>
  sv.alwaysff(posedge %clock) {
   sv.passign %reg, %c42 : i32
  } (asyncreset : posedge %reset) {
   sv.passign %reg, %c0 : i32
  }
}
```

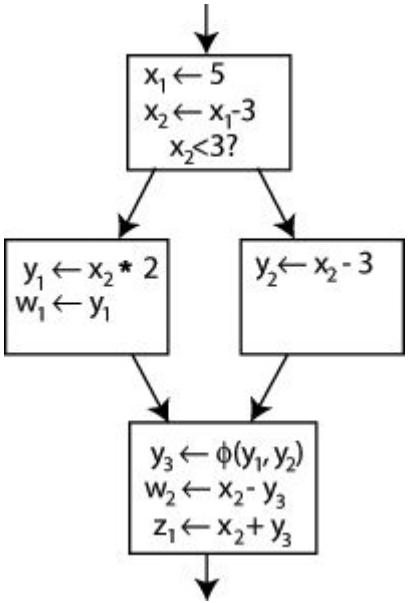
SystemVerilog

- Represents System Verilog syntax
- Designed for export, not transformation
- Readability and other syntactical improvements happen here
- Guides towards compatibility, but offers escape hatch

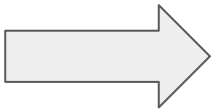
https://circt.llvm.org/docs/Dialects/SV/

Part 3: CIRCT challenges and opportunities

Challenge: graph regions in MLIR



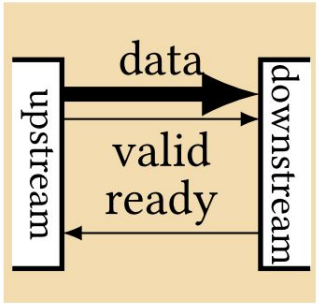
Traditional SSA Form



```
rtl.module @graph {
  %result0 = op0(%result1)
  %result1 = op1(%result0)
}
```

MLIR with non-SSA graph

Challenge: dynamic control logic

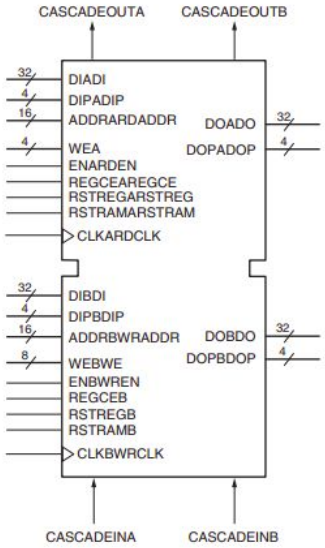


valid	ready	Meaning
1	0	Token valid; not consumed
1	1	Token transferred
0	–	No token to transfer

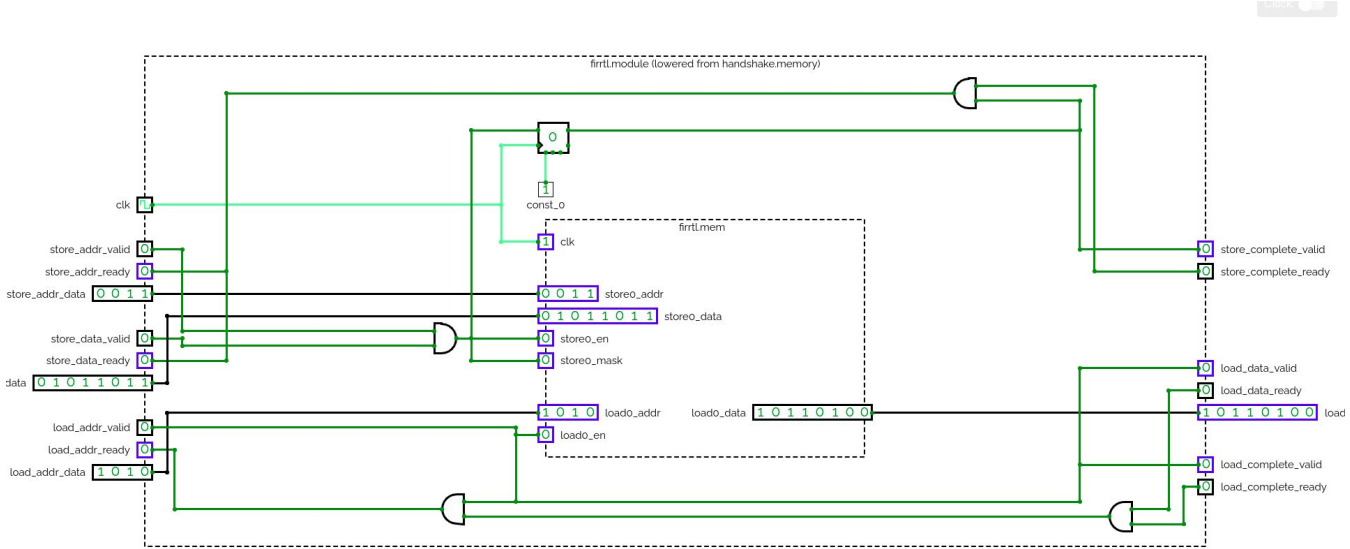
Signals for Valid / Ready handshake

Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. 2019. Compositional Dataflow Circuits.

Challenge: representing memory

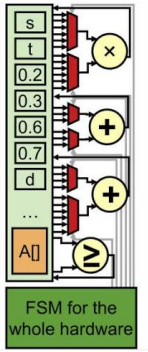


Xilinx BRAM

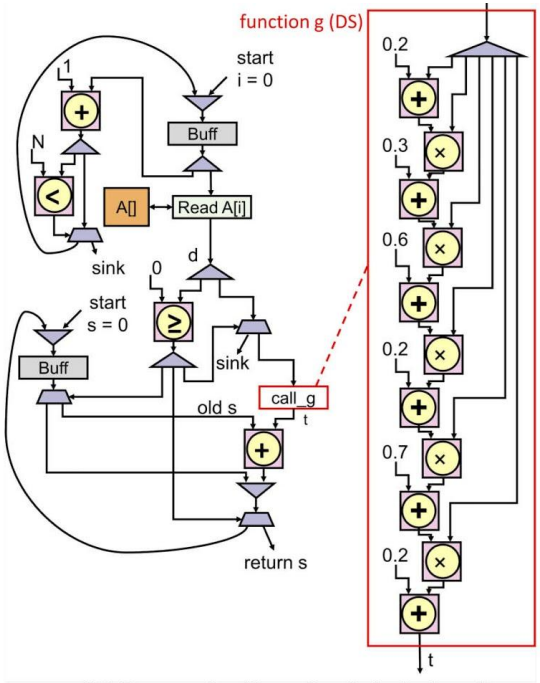


CIRCT example with FIRRTL memory

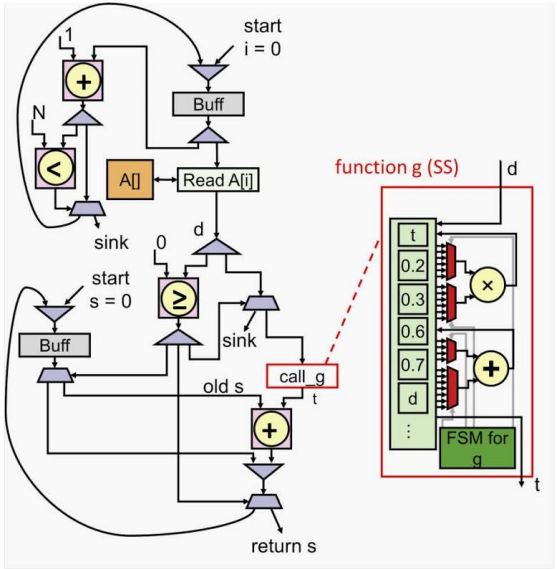
Challenge: handling dynamism while keeping static parts



(a) Statically scheduled circuit



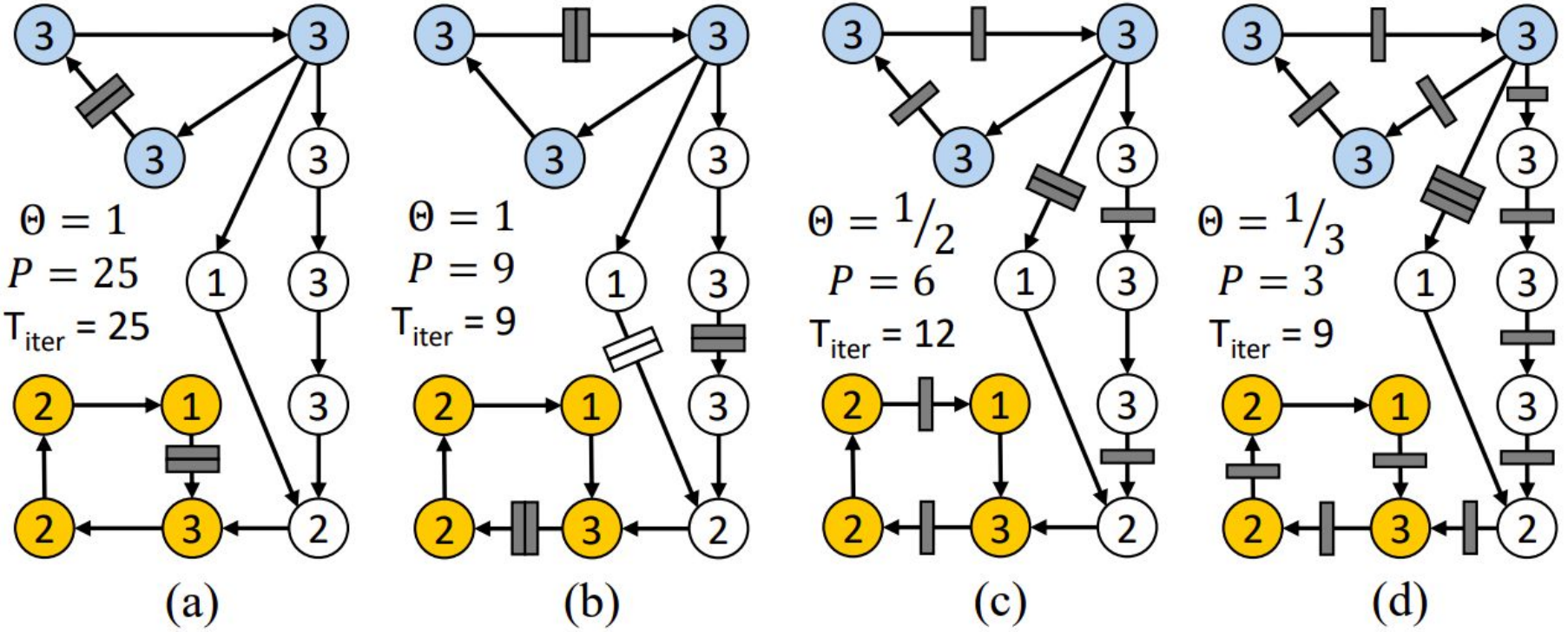
(b) Dynamically scheduled circuit



(c) Dynamically and statically scheduled circuit (our work)

Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-level Synthesis.

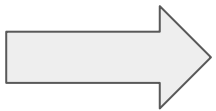
Opportunity: dynamic buffer optimization

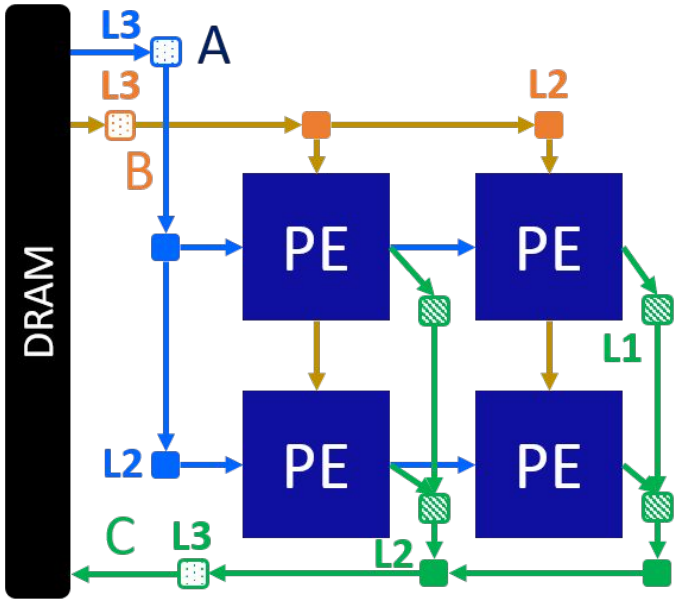


Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits.

Opportunity: static optimization and tiled architectures

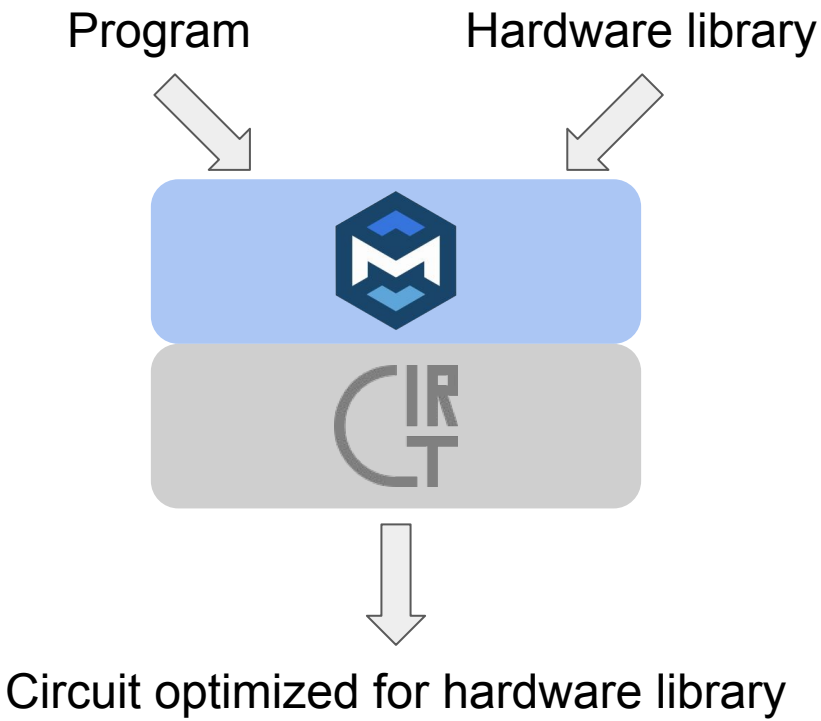
```
#pragma scop
for (int i = 0; i < 64; i++)
  for (int j = 0; j < 64; j++) {
    C[i][j] = 0;
    for (int k = 0; k < K64; k++)
      C[i][j] = C[i][j] + A[i][k] * B[j][k];
  }
#pragma endscop
```



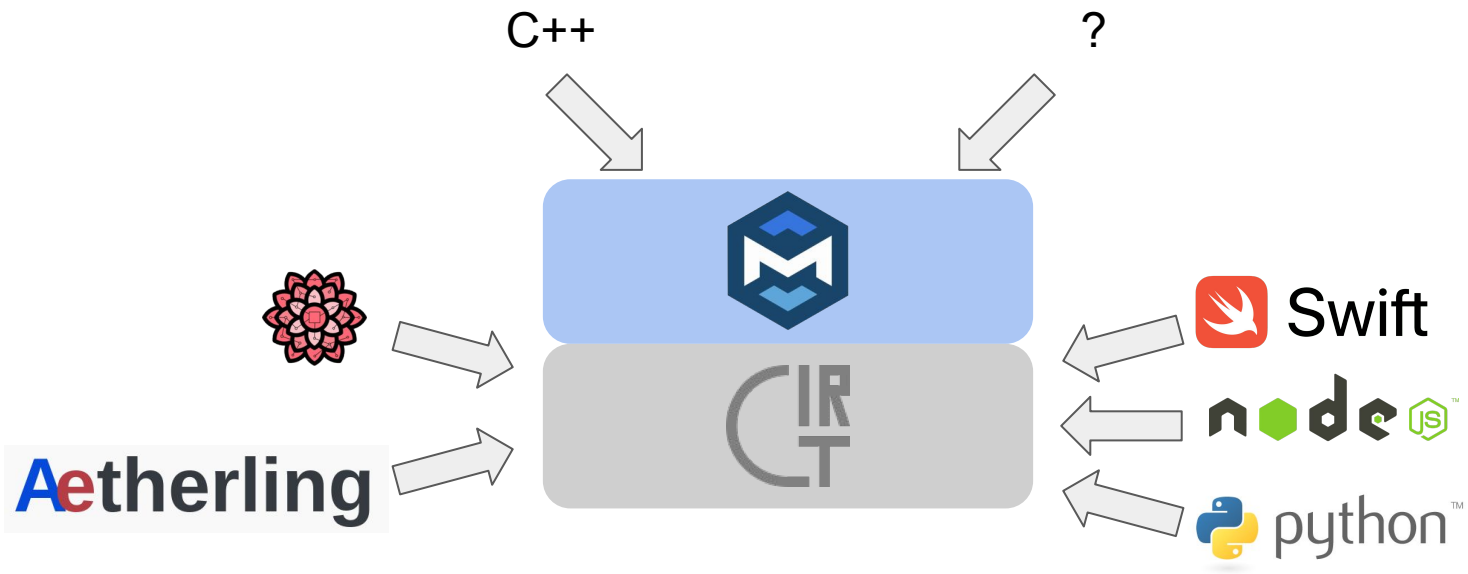


Jie Wang, Licheng Guo, and Jason Cong. 2021. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA.

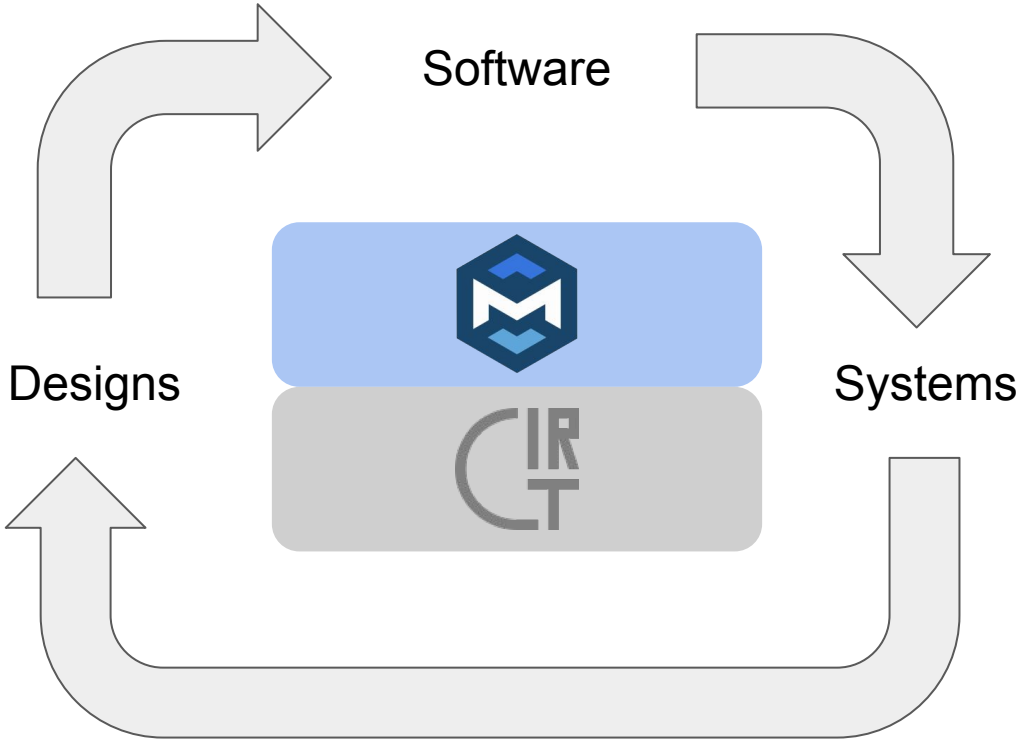
Opportunity: design space exploration



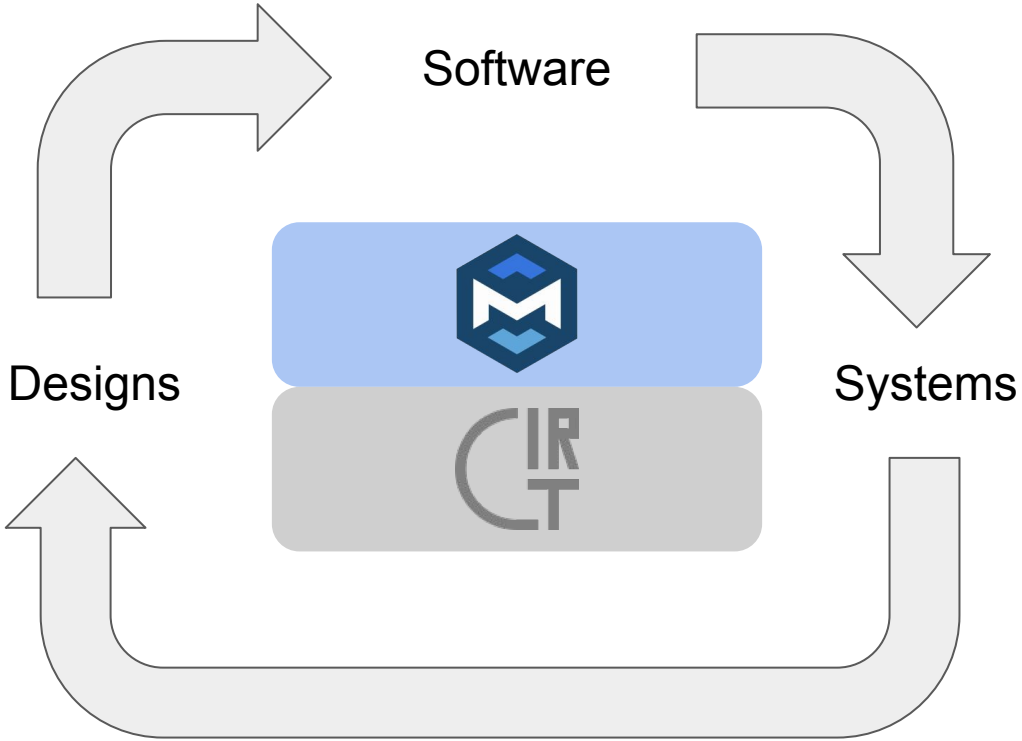
Opportunity: high-level languages



Opportunity: integrating software, systems, and designs



Conclusion



Acknowledgements

- SiFive: Chris Lattner, Schulyer Eldridge, Andrew Lenarth, Andrew Young
- Xilinx: Stephen Neuenndorffer, Hanchen Ye, Jianyi Chen
- Microsoft: John Demme

Getting Involved

- Website: circt.llvm.org
- GitHub: github.com/llvm/circt
- Forums: CIRCT on LLVM Discourse
- Chat: CIRCT on LLVM Discord

Questions?

mike@alloystack.io