AHS: An EDA Toolbox for Agile Chip Front-end Design

Tutorial @ ASPDAC 2025

Jan 20, 2025

https://ericlyun.me/tutorial-aspdac2025/

ASPDAC

Peking University

Team

Faculty

Yun (Eric) Liang
Professor
School of EECS/Integrated Circuit
Peking University
ericlyun@pku.edu.cn

Students

Youwei Xiao
3rd year Phd Student

Ruifan Xu
4th year Phd Student

Xiaochen Hao
5th year Phd Student

Zizhang Luo
3rd year Phd Student

Fan Cui
1st year Phd Student

Kexing Zhou
1st year Phd Student

Schedule

Time	Agenda	Presenter
50mins	Overview of AHS	Yun Liang
	Hands-on Session	
45mins	High-level Synthesis (ICCAD'22, FCCM'23, MICRO'24)	Ruifan Xu and Xiaochen Hao
20mins	Hardware Simulation (MICRO'23)	Kexing Zhou
45mins	Hardware Description Language (FPGA'24)	Youwei Xiao and Zizhang Luo
20mins	LLM-based Chip Design (ICCAD'24)	Fan Cui

Outline

- Overview
  - Hardware design background
  - Methodologies of AHS
- Hands-on Session

Hardware are Diverse

Processors / SoCs

XIANGSHAN

General-Purpose Accelerators

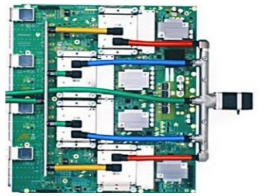
GPUs

VORTEX

Open-Source OpenCL Compatible RISC-V GPGPU

Domain-specific Accelerators

ML/DL



TPU V4

TVM-VTA

GEMMINI

Application-specific Accelerators

Example Applications:

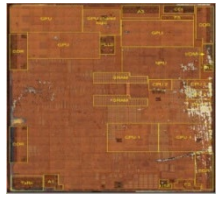
SpMV, SpMM, SLAM

Markov Chain Monte Carlo

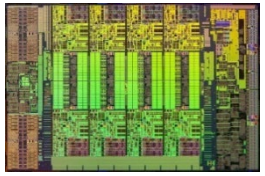
Xilinx Alveo

Amazon EC2

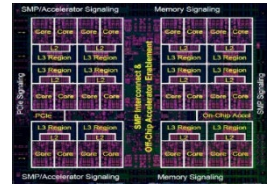
Chip Design Complexity



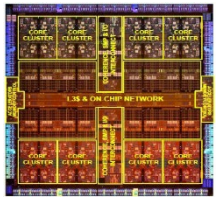
Apple A11
~4B transistors



Intel Haswell-EP Xeon E5
~7B transistors



IBM Power9
~8B transistors

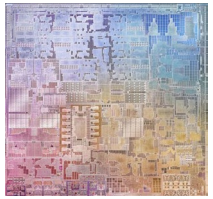


Oracle SPARC M7
~10B transistors

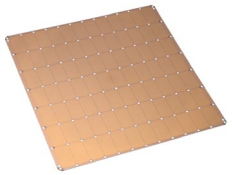
NVIDIA V100 Pascal
~21B transistors

Intel/Altera Stratix 10
~30B transistors

Xilinx VU9P
~ 35B transistors



Apple M1
~ 57B transistors

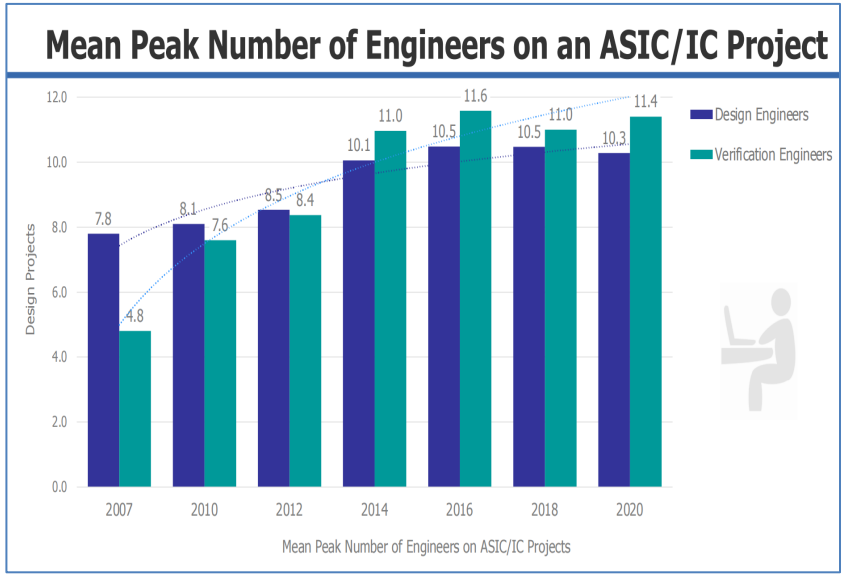


Cerebras WSE-2
~ 2.6T transistors

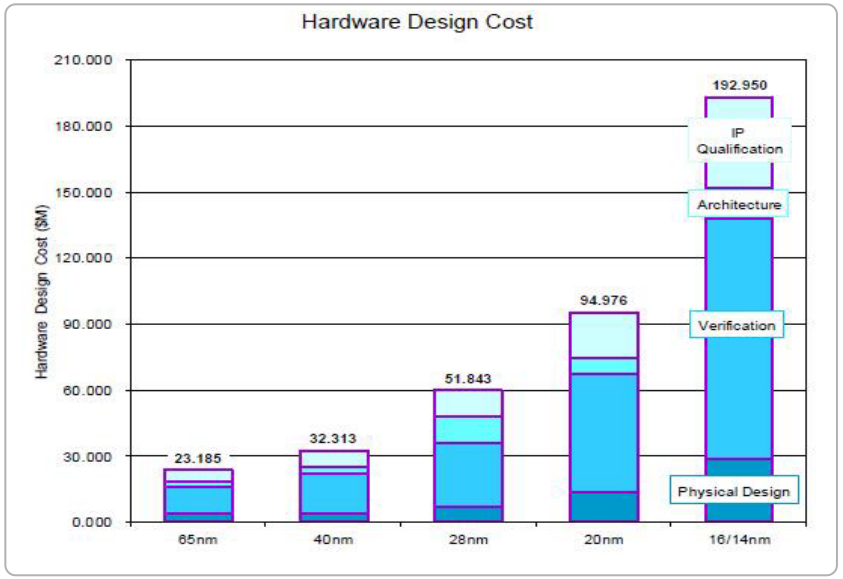
Verification is more Complex

- For hardware design, verification is necessary

Verification Engineers > Design Engineer



Verification Cost > Design Cost

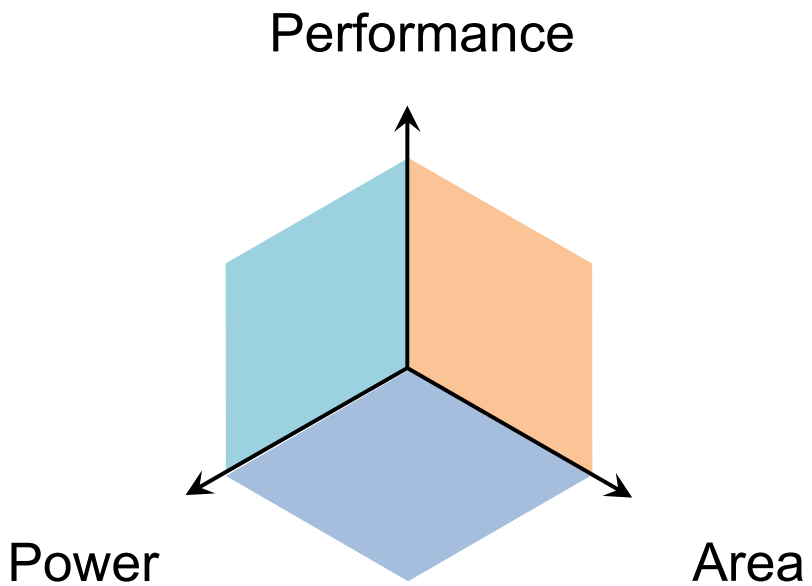


Hardware Simulation Hi-silicon case study

Case	Tech	Days
A	7nm	47 Days
B	7nm	61 Days
C	7nm	43 Days

Performance Metrics for Circuit Design

PPA



PPA + P(Productivity)

Productivity

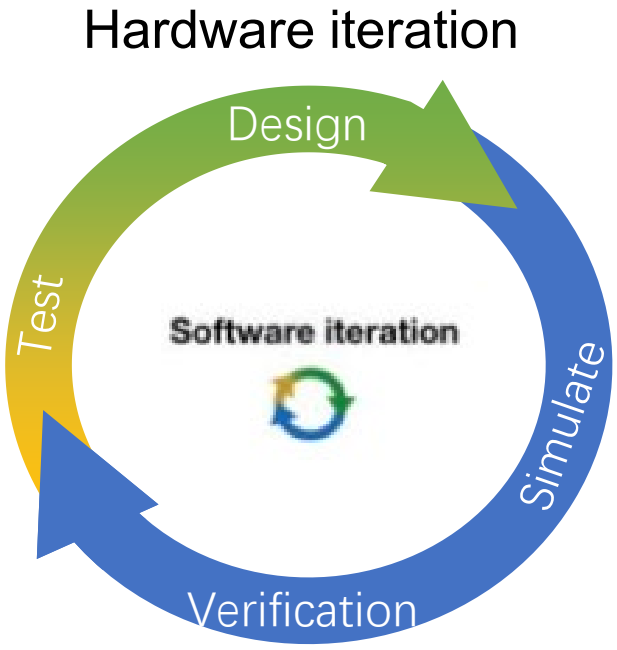
- Abstraction → Behavior level, RTL
- Program size → Line of code
- Verification → Simulation, Formal
- Agility → Reuse, start, iterated

Software Development

More: www.ibm.com/community/z/open-source-software/

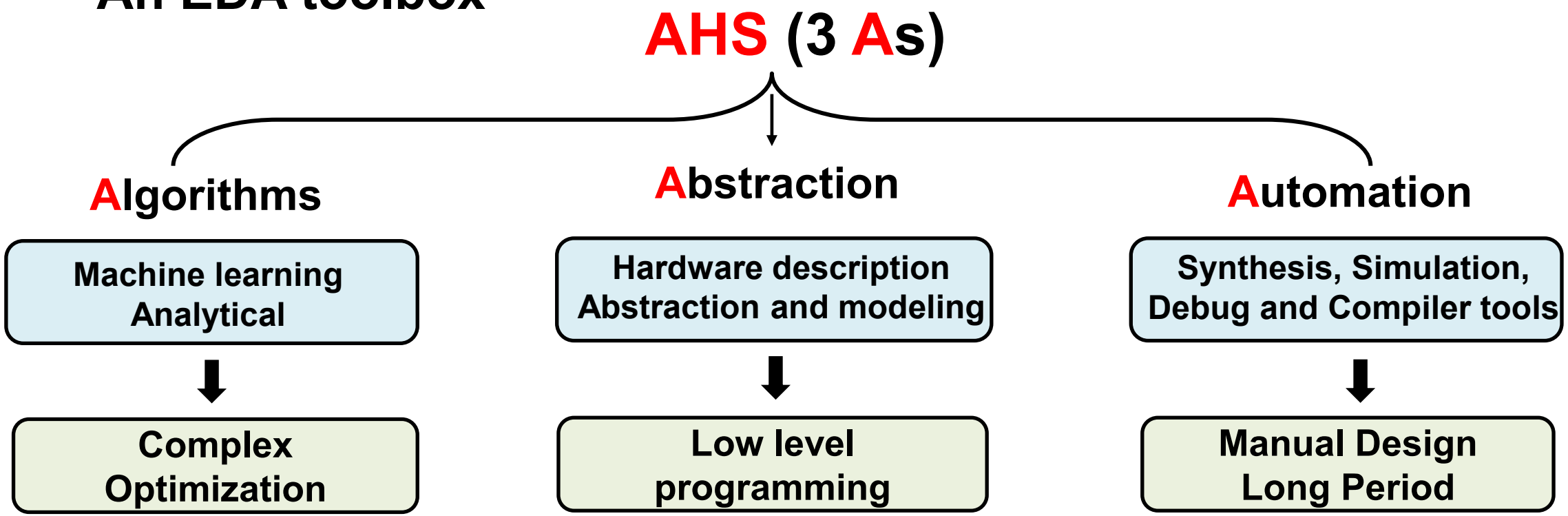
However, Hardware Development is Difficult

- Software
  - Open-source software ecosystem
  - Get projects started and iterated easily
- Hardware
  - Tools are seriously antiquated and lacking
  - Long design period and hard to debug

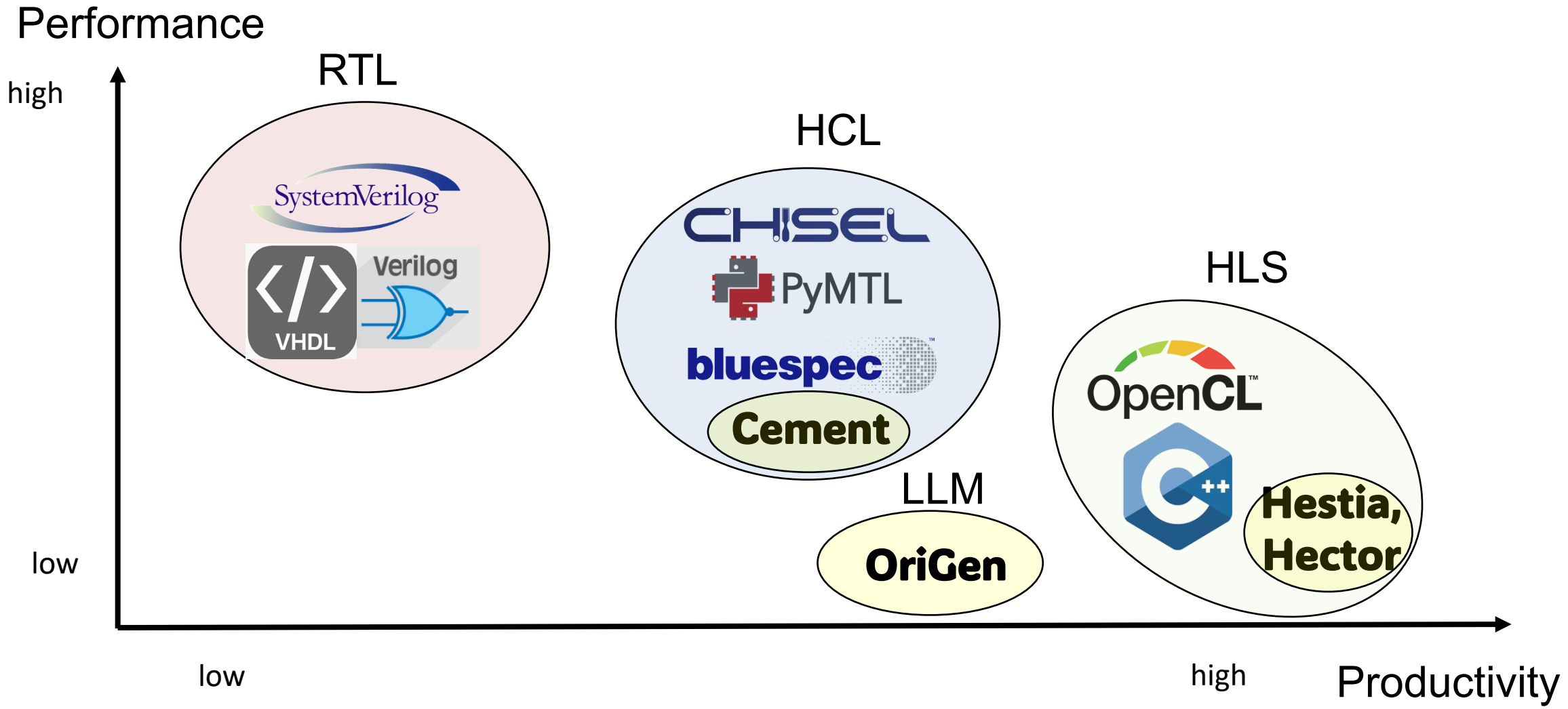


AHS: Agile Hardware Specialization

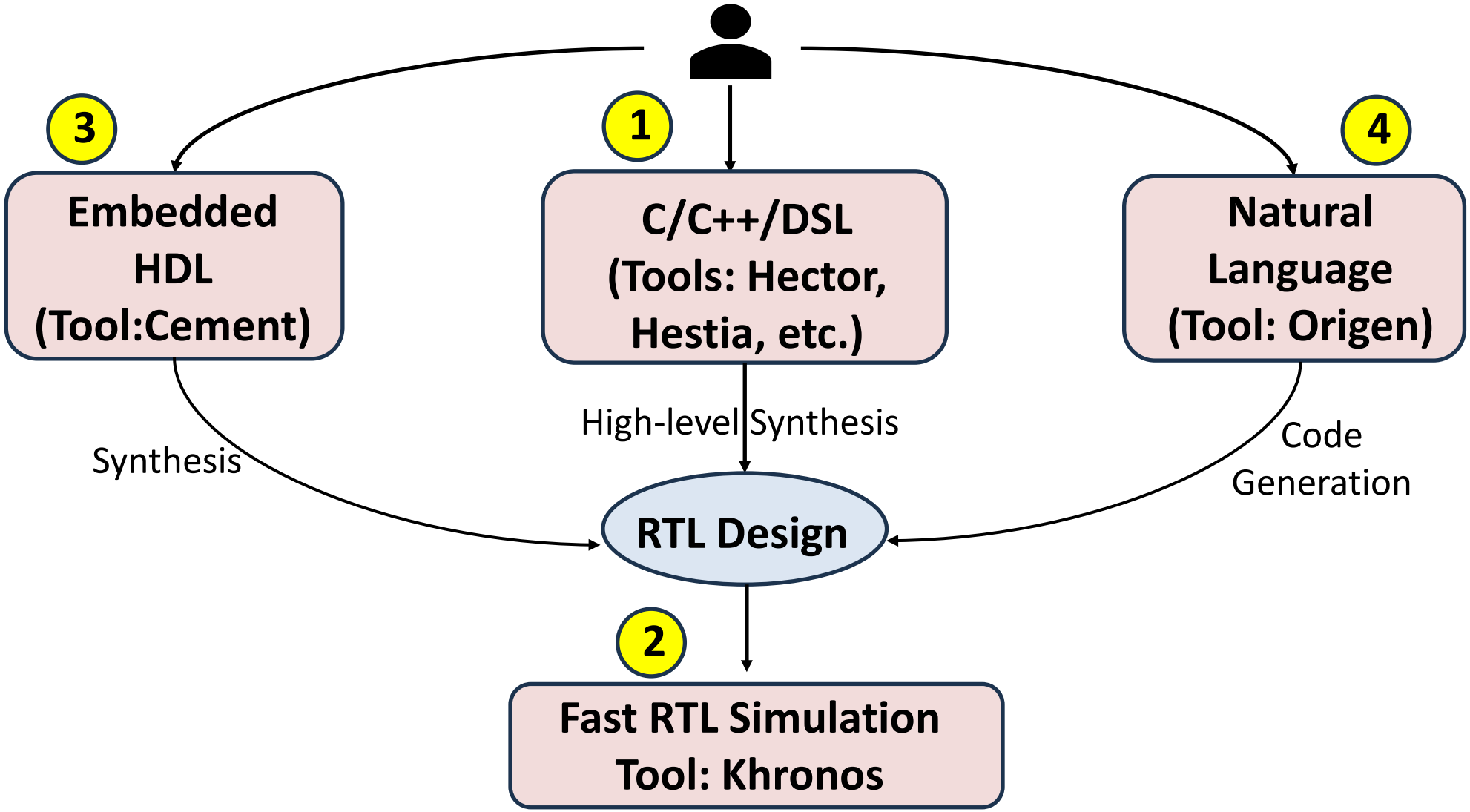
- **AHS**
  - **Design methodologies** for Agile chip design (front-end)
  - **An EDA toolbox**



Different Ways to Design Chip



Overview


3
1
4
Embedded HDL (Tool:Cement)
C/C++/DSL (Tools: Hector, Hestia, etc.)
Natural Language (Tool: Origen)
Synthesis
High-level Synthesis
Code Generation
RTL Design
2
Fast RTL Simulation Tool: Khronos

AHS Resource

- Webpage: https://ericlyun.me/tutorial-aspdac2025/
  - Papers, presentation, code

- High-level Synthesis and DSL
  - ICCAD'22, FCCM'23, MICRO'24
- Hardware Simulation/Verification
  - MICRO'23
- Embedded Hardware Description Language
  - FPGA'24
- LLM-assisted RTL generation
  - ICCAD'24

High-level Synthesis

- High-level synthesis (HLS) allows the designers to design hardware at a high-level abstraction

HLS tools

LegUp

Catapult

DYNAMATIC

XILINX VITIS

cadence

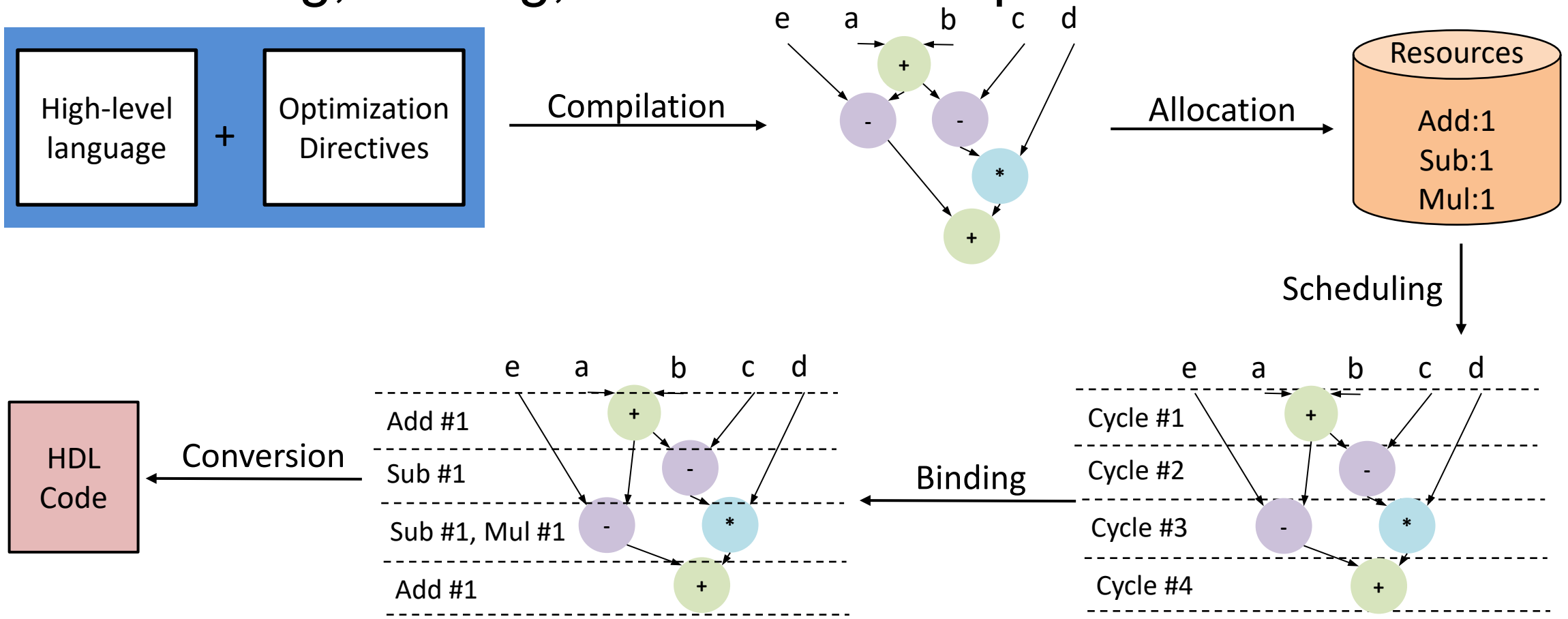
Applications

Google VCU

Vitis AI

A Typical HLS Flow

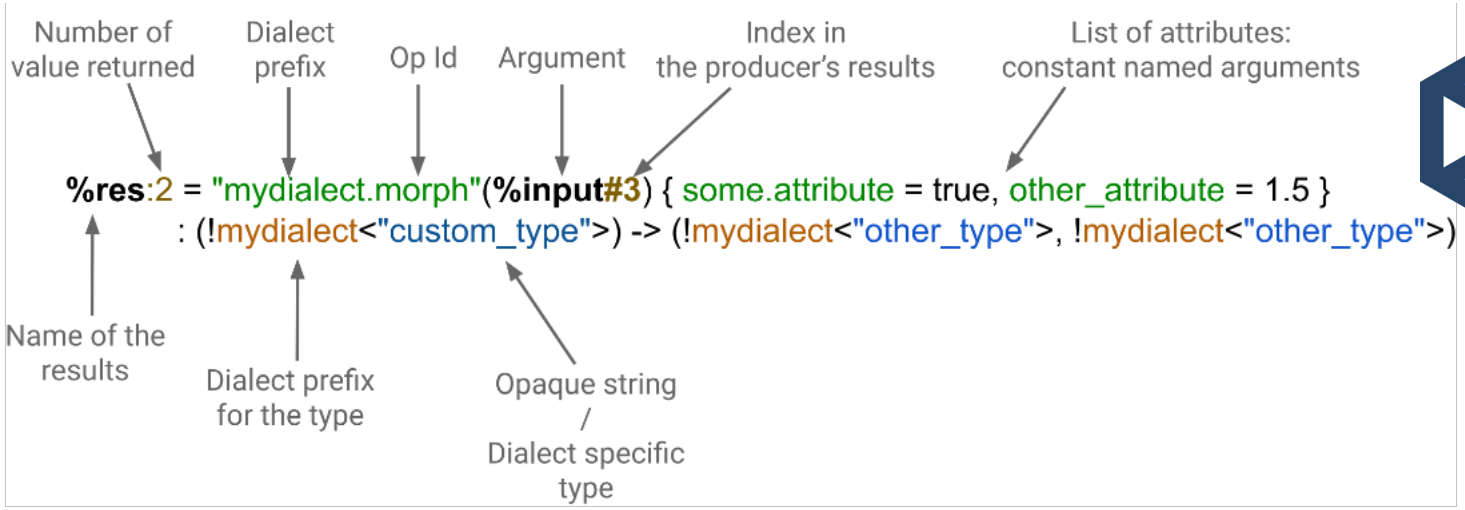
- HLS is a complex procedure including allocation, scheduling, binding, and additional optimizations



MLIR Infrastructure

- A novel compiler infrastructure that greatly facilitates the implementation of user-defined IRs and transformations
  - Reuse IR and extend new IRs
  - Provides a generic form of operations

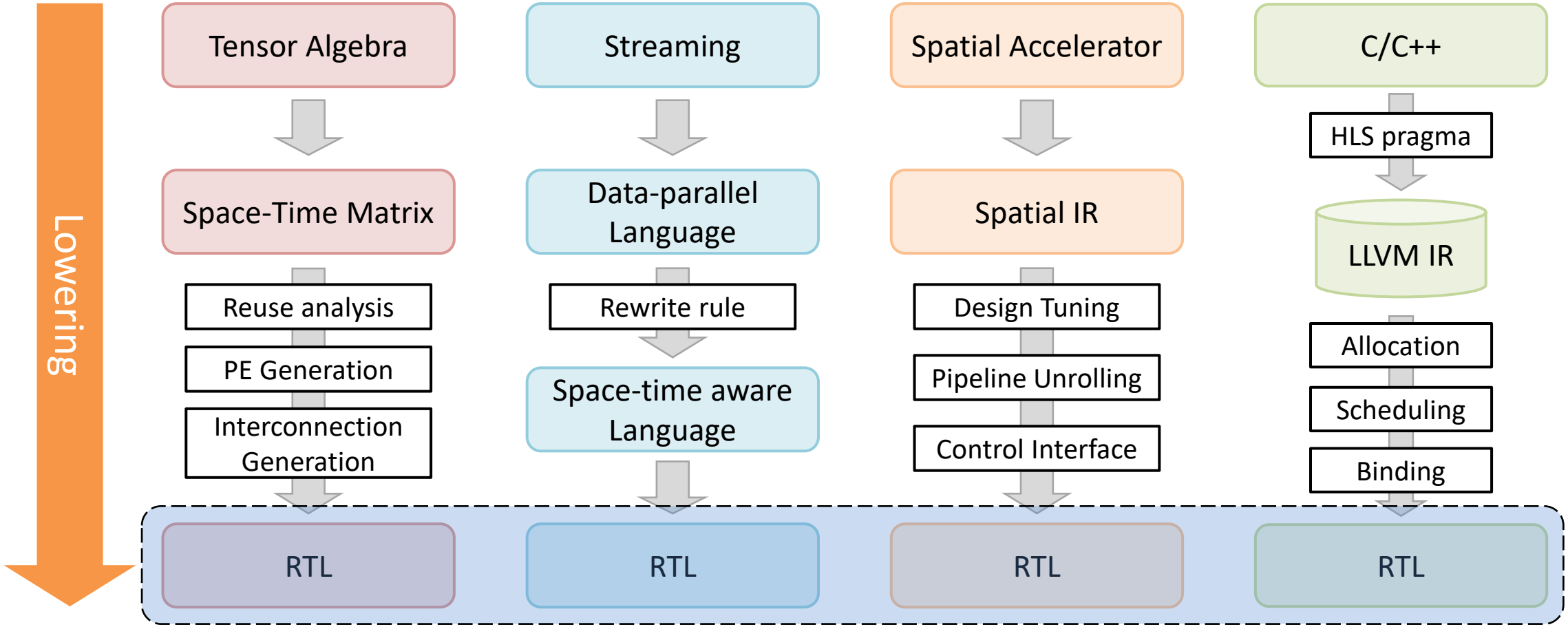
MLIR generic Representation



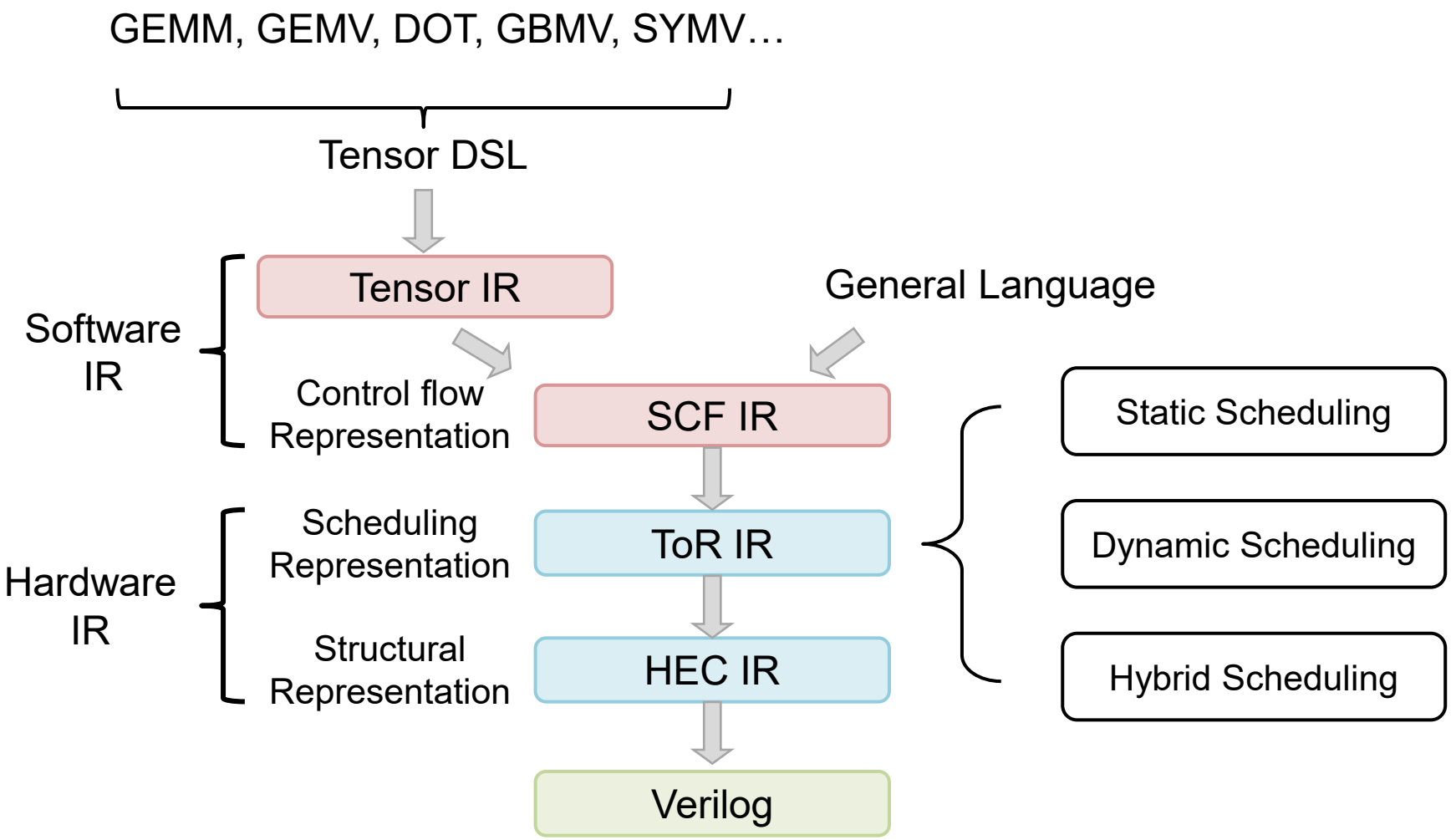
Widely used in recent compilers

Hardware Generator vs. General Flow

- Extend general HLS flow with new IRs for specific domains

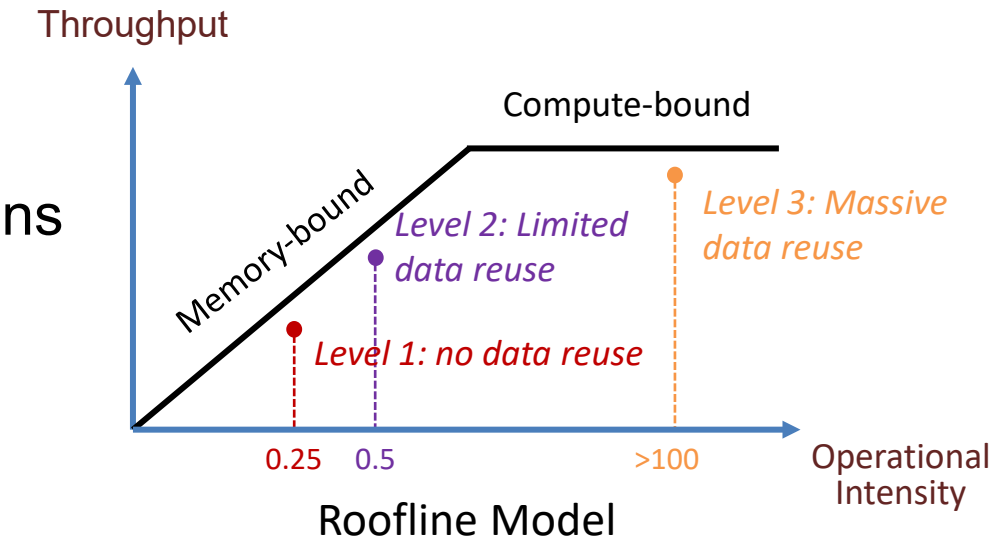


Overview of Hector



Tensor Specialization

- Tensor computations are ubiquitous
  - Machine learning, scientific computation, etc.
- BLAS (Basic Linear Algebra Subprograms)
  - Level 1: scalar, vector, vector-vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations
- Requiring substantial optimizations
  - Exploiting parallelism and data reuse



Roofline Model

Tensor DSL

- Tensor DSL and IRs
  - Uniform recurrence equations (UREs) and space-time transformation

Matrix Multiply

$$C_{ij} = \sum_k a_{ik} b_{kj}$$

A DSL embedded in C++

What to compute (UREs)

```
A(k, j, i) = select(j == 0, a(k, i), A(k, j-1, i));
B(k, j, i) = select(i == 0, b(j, k), B(k, j , i-1));
C(k, j, i) = select(k == 0, 0, C(k-1, j, i))
                            + A(k, j, i) * B(k, j, i);
c(   j, i) = select(k ==K-1, C(k, j, i));
```

Build a loop nest

```
A.merge_ures(B, C, c)
A.set_bounds(i, 0, I, …)
A.tile({i, j, k}, {ii, jj, kk}, {iii, jjj, kkk})
```

Mapping

```
A.space_time_transform(iii, jjj)
```

Data movement

```
Stensor DA(DRAM), SA(SRAM), …
A >> DA.out(kkk)
  >> SA.scope(k).out(iii, kkk);
```

Target

```
c.compile_to_device(“matmul.mlir”, Target::MLIR);
```

Tensor IR

Tensor IR

- Loops and C-like statements
- High-level loop transformation optimizations

SCF IR

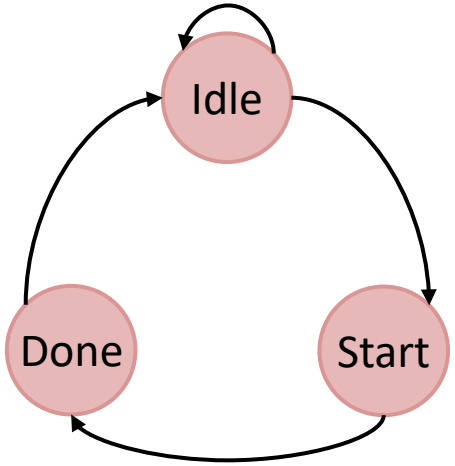
- Affine expressions and SSA statements
- Low-level basic block optimizations

```
for (C.s0.i, 0, 16) {
 for (C.s0.j, 0, 16) {
  allocate sum[float32 * 1]
  produce sum {
   sum[0] = 0.000000f
   for (sum.s1.k$x, 0, 16) {
     sum[0] = sum[0] + (image_load("A",…)
*image_load("B",…))
   }
  }
  consume sum {
   image_store("C", …, sum[0])
  }
 }
}
```

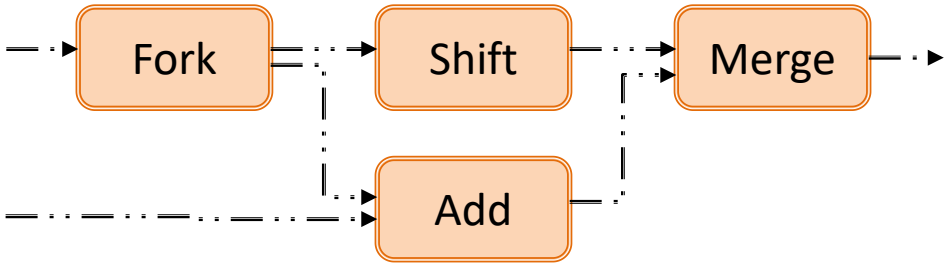
Lower

```
affine.for %arg0 = 0 to 16 {
  affine.for %arg1 = 0 to 16 {
    %alloc_2 = memref.alloc()
     memref.store %c0_i32, %alloc_2[%c0]
    affine.for %arg2 = 0 to 16 {
      %1 = affine.load %alloc_0[%arg1, %arg2]
      %2 = affine.load %alloc[%arg2, %arg0]
      %3 = arith.muli %2, %1
      %4 = memref.load %alloc_2[%c0]
      %5 = arith.addi %4, %3
      memref.store %5, %alloc_2[%c0]
    }
    %0 = memref.load %alloc_2[%c0]
    affine.store %0, %alloc_1[%arg1, %arg0]
  }
}
```

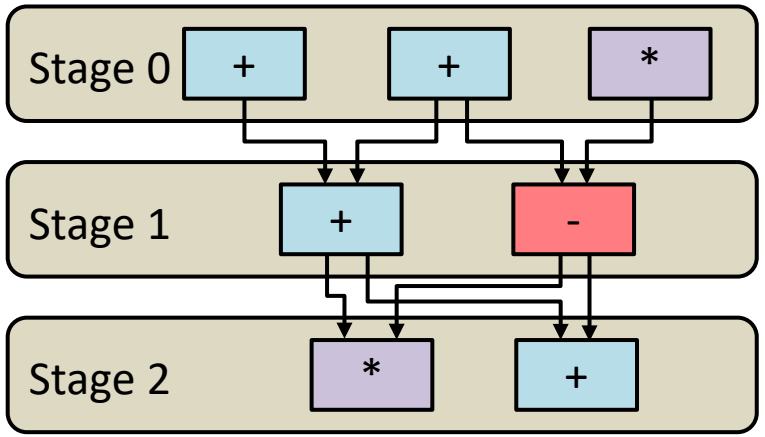
Hardware IR



Finite State Machine



Latency-insensitive Design



Pipeline

Hardware IR

TOR IR

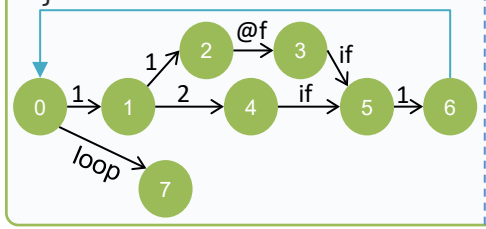
- High-level IR
- Software-like computation with high-level schedule graph

HEC IR

- Low-level IR
- Unified description through allocate-assign mechanism

(a) Schedule IR

```
tor.topo (0 to 7) {
  tor.from 0 to 1 "seq:1"
  tor.from 1 to 2 "seq:1"
  tor.from 2 to 3 "call"
  tor.from 1 to 4 "seq:2"
  tor.from 3, 4 to 5 "if"
  tor.from 5 to 6 "seq:1"
  tor.from 0 to 7 "for"
}
```



(b) Functional IR

```
tor.for %i = %c0 to %c10 step %c1 {
  %m = tor.load %mask[%i] on (0 to 1)
  %a = tor.if %m then {
    %x = tor.addi %i %c1 on (1 to 2)
    %y = tor.subi %i %c1 on (1 to 2)
    %fx = tor.call @f(%x, %y) on (2 to 3)
    tor.yield %fx
  } else {
    %ii = tor.muli %i %i on (1 to 4)
    tor.yield %ii
  } on (1 to 5)
  tor.store %a to %A[%i] on (5 to 6)
} on (0 to 7)
```

(a) allocation

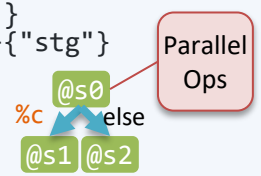
```
%m.lhs, %m.rhs, %m.res=
  hec.primitive "muli": i32
%i=hec.wire "i": i32
```

(b) assign

```
%0=add %cst1 %i
hec.assign %m.lhs=%0
hec.assign %m.rhs=%i
```

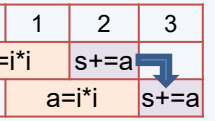
```
component @STG {
 // allocations
 stateset {
  state @s0 {
   // assigns
   transition {
    goto @s1 if %c
    goto @s2 //else
   }
  } //other states
 }
}{"stg"}
```

Parallel Ops



(c) STG

```
component @Pipe {
 // allocations
 stageset {
  stage @s0 {
   // assigns
  } // other stages
  stage @sN {
   // assigns
   deliver %x to %y
  }
 }
}{"pipeline",II=1}
```



(d) Pipeline

```
component @Hs {
 // instances
 %f.a,r =
  instance.@f
 // elastic units
 %m.i1,i2,o =
  primitive."merge"
 graph {
  assign %m.i1=%f.r
  assign %x=%m.o
 }
}{"handshake"}
```

(e) Handshake

Tensor Accelerator Generation

Level	Kernel	Name	Compute	Frequency	LUTs	DSPs	Throughputs	Speed up
Level 3	GEMM	matrix-matrix multiply	$C = \alpha AB + \beta C$	244 Mhz	49%	86%	620 GFlops	-
	SYMM	symmetric matrix-matrix multiply	$C = \alpha AB + \beta C, A = A^T$	244 Mhz	49%	86%	620 GFlops	-
	HEMM	hermitian matrix-matrix multiply	$C = \alpha AB + \beta C, A = A^H$	230 MHz	41%	86%	582 GFlops	-
	SYRK	symmetric rank-k update to a matrix	$C = \alpha AA^T + \beta C$	259 Mhz	43%	68%	513 GFlops	1.93X
	HERK	hermitian rank-k update to a matrix	$C = \alpha AA^H + \beta C$	228 Mhz	35%	68%	459 GFlops	1.96X
	SYR2K	symmetric rank-2k update to a matrix	$C = \alpha AB^T + \alpha BA^T + \beta C$	253 Mhz	48%	68%	476 GFlops	1.81X
	HER2K	hermitian rank-2k update to a matrix	$C = \alpha AB^H + \alpha BA^H + \beta C$	252 Mhz	42%	68%	426 GFlops	1.63X
	TRMM	triangular matrix-matrix multiply	$B = \alpha AB$	238 Mhz	44%	68%	471 GFlops	1.93X
Level 2	GEMV	matrix-vector multiply	$y = \alpha Ax + \beta y$	282 Mhz	20%	2%	16 GFlops	-
	GBMV	banded matrix-vector multiply	$y = \alpha Ax + \beta y$	277 Mhz	21%	2%	16 GFlops	7.35X
	SYMV	symmetric matrix-vector multiply	$y = \alpha Ax + \beta y$	267 Mhz	39%	4%	15 GFlops	1.79X
	TRMV	triangular matrix-vector multiply	$x = Ax$	254 Mhz	23%	2%	15 GFlops	1.75X
	GER	performs the rank 1 operation	$A = \alpha xy^T + A$	259 Mhz	20%	1%	7.6 GFlops	-
Level 1	DOT	dot product	$dot = xy$	308 Mhz	17%	1%	8 GFlops	-

Using 20-30 lines to achieve performance comparable to ~1000 lines of manual HLS design

Experiment Results

- Comparison against static and dynamic HLS tools

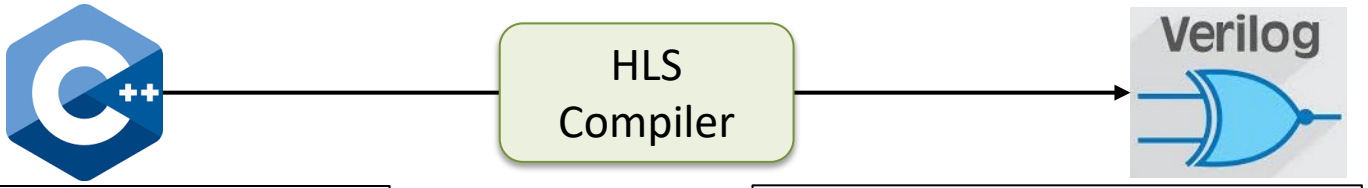
Benchmark	LUTs		FFs		Cycles (k)		Period (ns)	
	Vitis	Ours	Vitis	Ours	Vitis	Ours	Vitis	Ours
GEMM	852	890	1958	1600	3923	3752	5.073	4.140
Stencil2D	94	192	188	370	320	313	4.545	3.904
Stencil3D	454	372	668	890	103	104	5.692	4.672
SPMV (CSR)	881	932	1934	1625	37.1	34.2	5.299	4.848

Benchmark	LUTs		FFs		Cycles (k)		Period (ns)	
	DYN	Ours	DYN	Ours	DYN	Ours	DYN	Ours
AEloss Pull	331	280	265	212	12.5	14.7	6.1	5.6
AEloss Push	1118	250	900	199	326	294	6.2	5.5
Stencil2D	1626	1227	1379	891	430	399	7.3	6.6

Comparable result with existing HLS tools

Semantic Gap between Software and RTL

- The large gap necessities the verification of HLS design

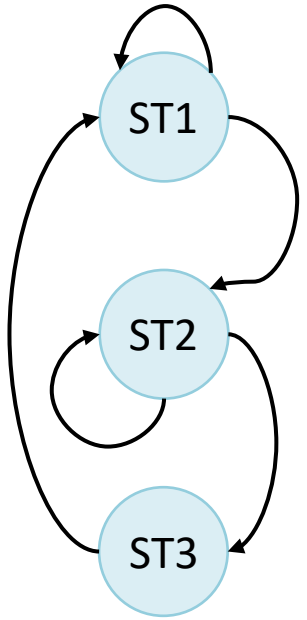



```
uint64_t crc32_context=0xFFFFFFFFUL;
void crc32_byte(uint64_t b) {
  crc32_context =
    ((crc32_context>>8)&0x00FFFFFF) ^
    crc32_tab[(crc32_context^b)&0xFF];
}
void crc32_8bytes (uint64_t val) {
  for (int i=0; i<8; i++) {
#pragma HLS UNROLL
    crc32_byte((val>>(i<<3))&0xff);
  }
}
uint64_t result() {
#pragma HLS loop_merge
  for (int i=0; i<2; i++)
    crc32_8bytes(A[i]);
  for (int i=0; i<2; i++)
    crc32_8bytes(B[i]);
  return crc32_context;
}
```

10X more lines of code

CSIM: 73FBD3D2
Mis-match
RTL : E01BF9AD

```
always @ (*) begin
  case (CS)
    ST_state1 : begin
      if (((1'b1 == CS_state1)
          & (ap_start == 1'b1))) begin
        ap_NS_fsm = ST_state2;
      end else begin
        ap_NS_fsm = ST_state1;
      end
    end
    ST_state2 : begin
      if (((1'b1 == CS_state2)
          & (cond1_48 == 1'd1))) begin
        ap_NS_fsm = ST_state3;
      end else begin
        ap_NS_fsm = ST_state2;
      end
    end
  endcase
end
```



Existing HLS tools are unreliable, sometimes generating wrong hardware

Yann Herklotz "Formal Verification of High-Level Synthesis" OOPSLA 2021

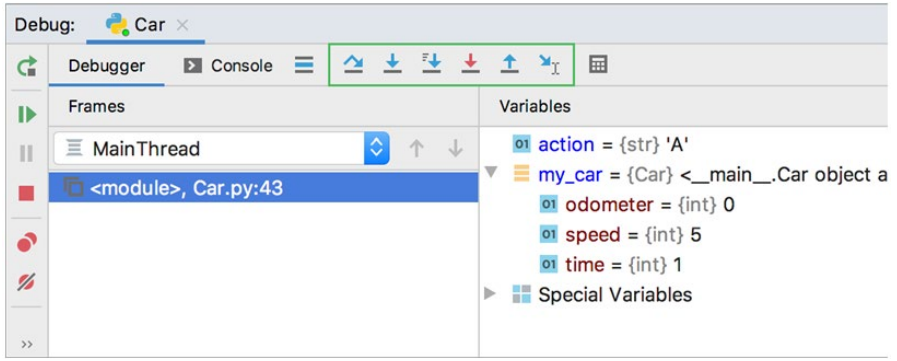
Debugging HLS design

- Existing HLS tools have limited support for debugging

Software Debugging

Single Stepping

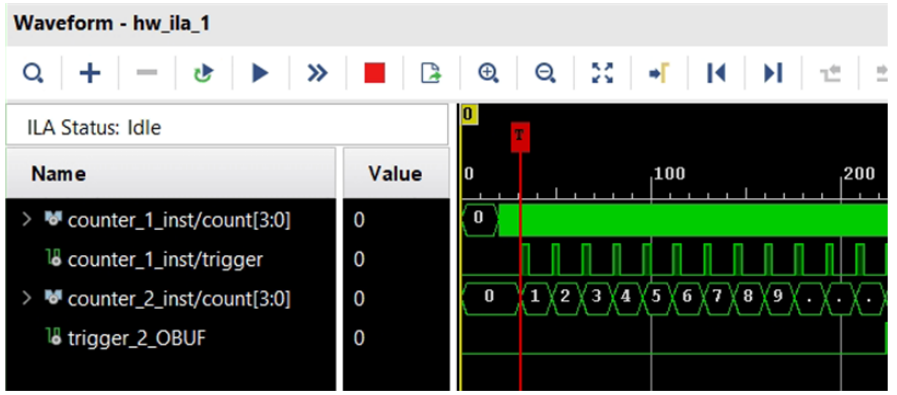
Interactive Debugging



RTL Debugging

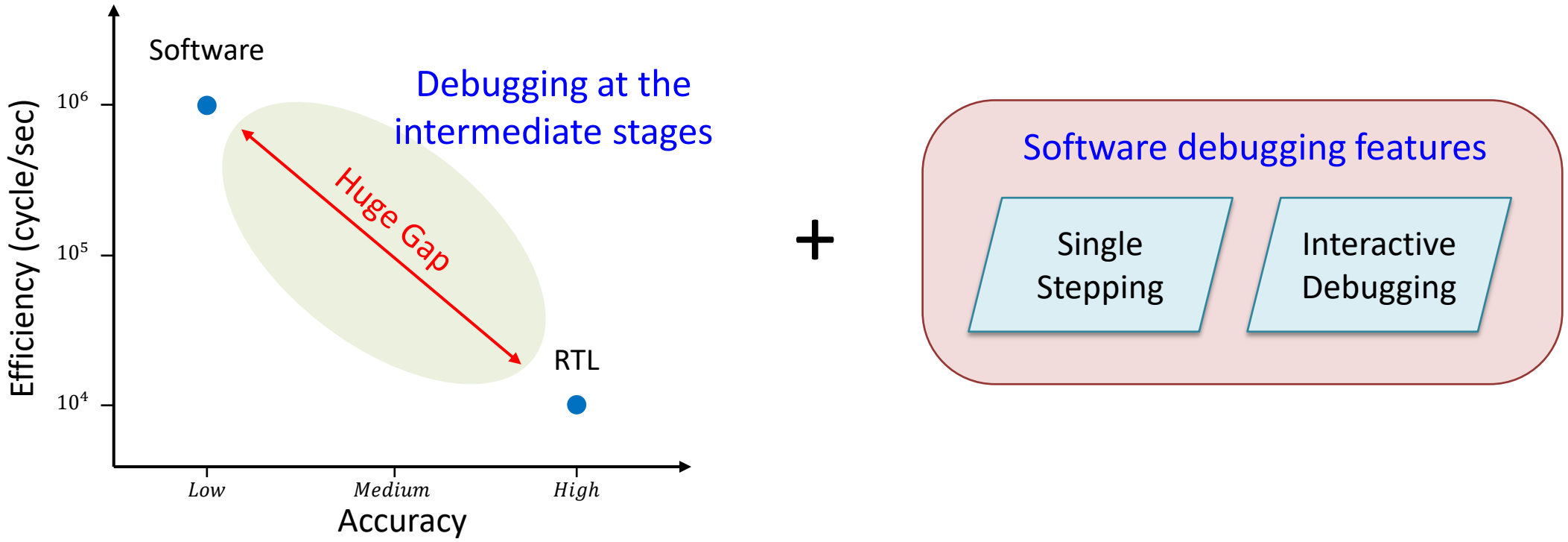
Waveform analysis

Monitor & logging



Key Idea

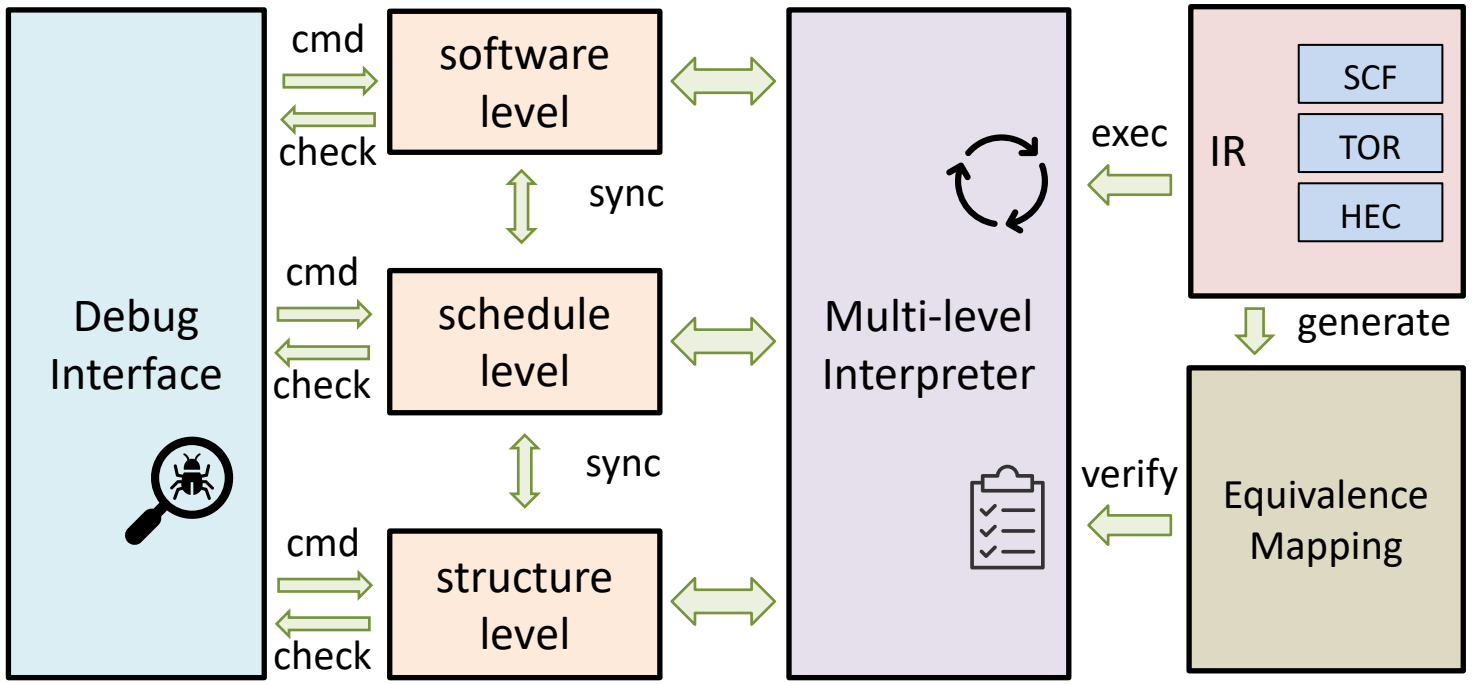
- Debugging at intermediate stages can get a better trade-off between efficiency and accuracy



Overview of Hestia

- An efficient cross-level debugger for HLS designs that enables breakpoints and stepping at multiple granularity

Breakpoint
Watchpoint
Step
Continue
Switch
Load
……



Experiment Results

- Comparison of simulation efficiency against RTL level

Benchmark	Software (sec)	Schedule (sec)	Error (%)	Structure (sec)	RTL (sec)	Cycle (k)
GEMM	0.46	1.88	0.109	14.22	119.31	3748.0
Stencil2D	0.34	0.53	0.000	1.45	17.04	312.9
Stencil3D	0.16	0.23	0.001	1.57	11.3	103.6
SPMV (CSR)	0.01	0.02	1.442	0.17	8.94	34.2
AelossPull	0.00	0.03	0.000	0.44	12.51	15.4
AelossPush	0.70	1.98	0.006	26.88	71.95	1502.7

Improve by 174X and 19X on average compared to RTL simulator

AHS Resource

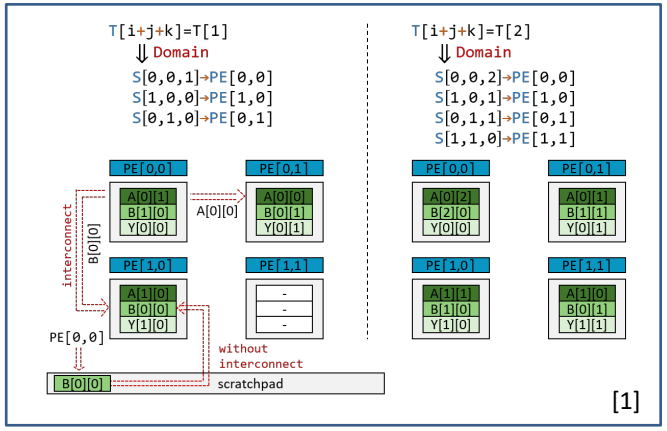
- Webpage: https://ericlyun.me/tutorial-aspdac2025/
  - Papers, presentation, code

- High-level Synthesis and DSL
  - ICCAD'22, FCCM'23, MICRO'24
- Hardware Simulation/Verification
  - MICRO'23
- Embedded Hardware Description Language
  - FPGA'24
- LLM-assisted RTL generation
  - ICCAD'24

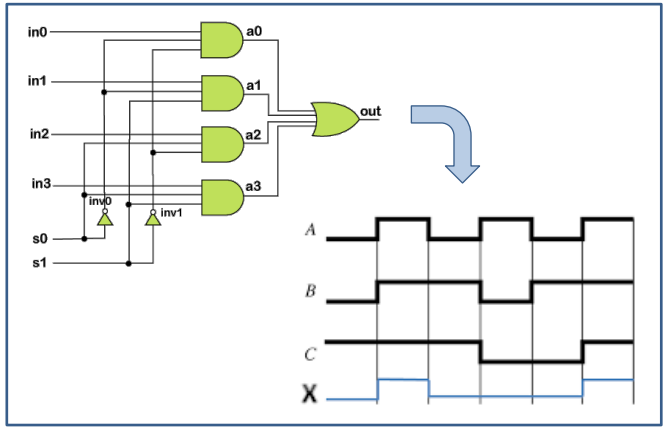
RTL Simulation

- RTL simulation is an important tool in HW flow
  - RTL sim. is cycle-accurate
  - Upstream tasks rely heavily on RTL simulation

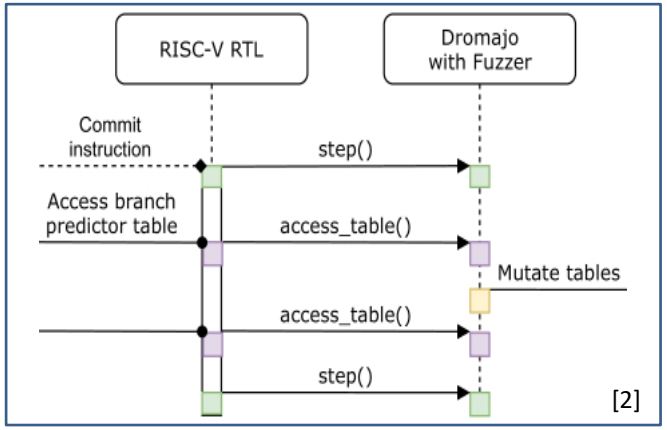
Design Space Exploration



Functional Verification & Coverage



Co-Simulation



[1] Image from TENET, TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation, ISCA'21
[2] Image from Dromajo, Effective Processor Verification with Logic Fuzzer Enhanced Co-simulation, MICRO'21

SW RTL Simulation is Slow

- SW RTL simulation is very slow on large design
  - Simulation only 100~1000 cycle/s
  - Frequency of MHz or GHz in real chip

black-parrot Core

434 cycle/s

26 day/Gcycle

XuanTie C910 Core

457 cycle/s

25 day/Gcycle

XiangShan Core

563 cycle/s

20 day/Gcycle

Vortex GPU Core

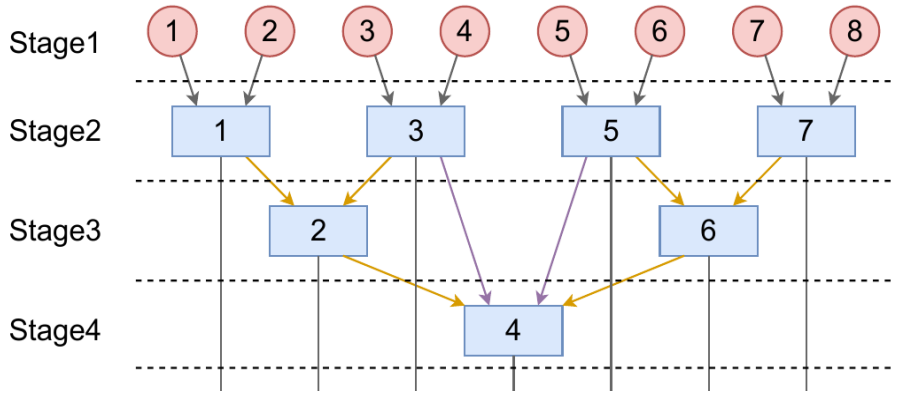
81 cycle/s

5 month/Gcycle

Rresult is conducted by opensource RTL simulator, Verilator

Memory Access is the Bottleneck

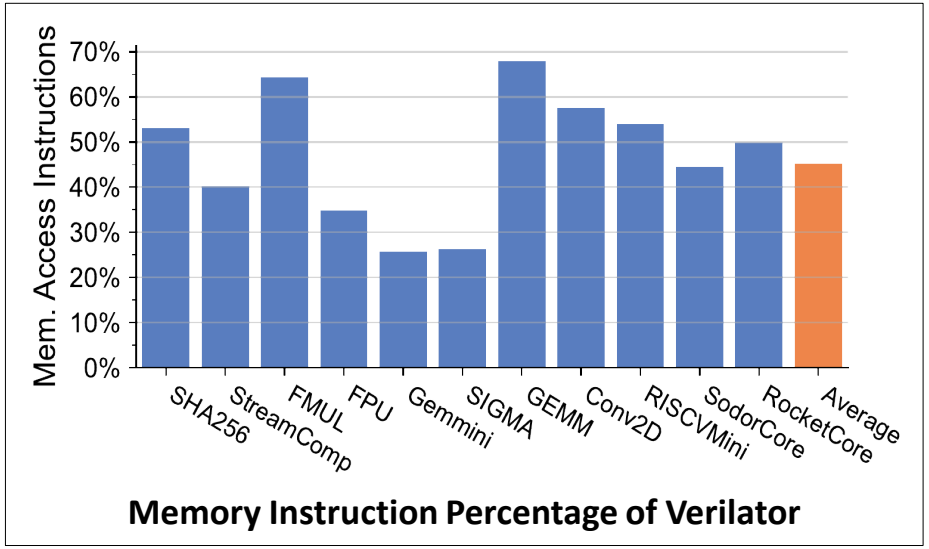
- Memory access
  - Verilator spend **~45%** instruction to access memory
  - Large amount of **reg buffers** are in HW design



Large Amount of Register Buffers in SIGMA[1]

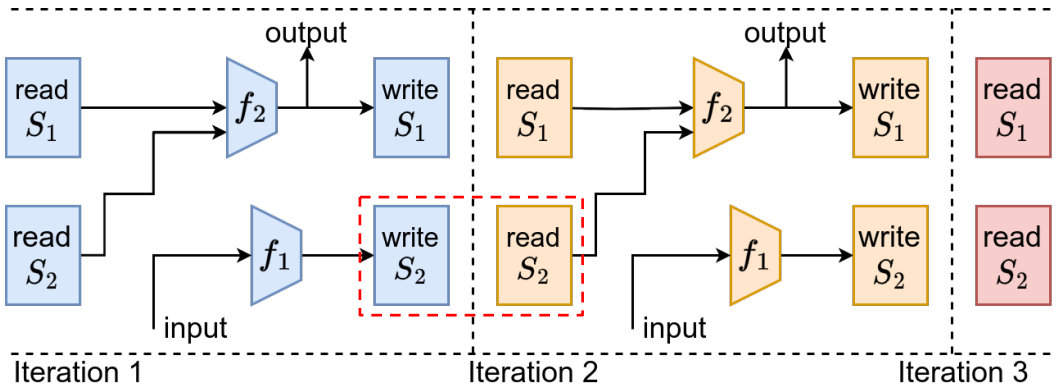
- Prior works ignore the optimization of memory access in RTL simulation

Simulator	Key Feature	Behavior
Verilator	FC simulator	Memory Access
ESSENT[1]	Mix event with FC	More buffer
RepCut[2]	Multi-threading	Thread

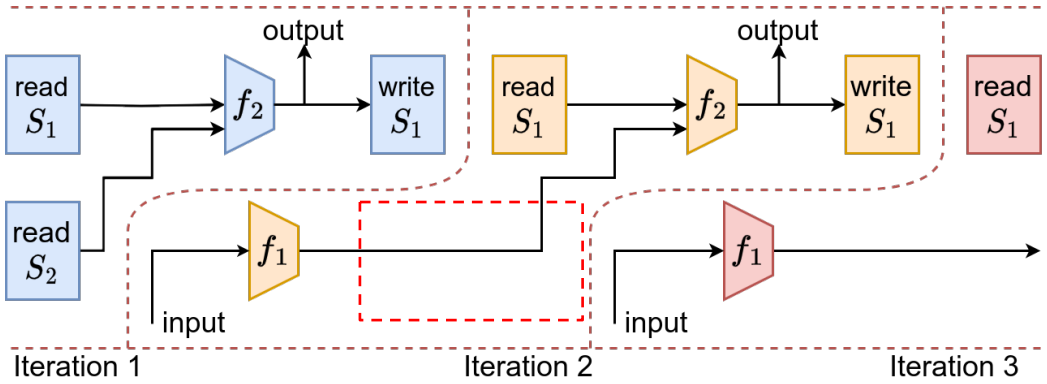


Memory Instruction Percentage of Verilator

Eliminate Memory Access by Rescheduling



Full Cycle RTL Simulation

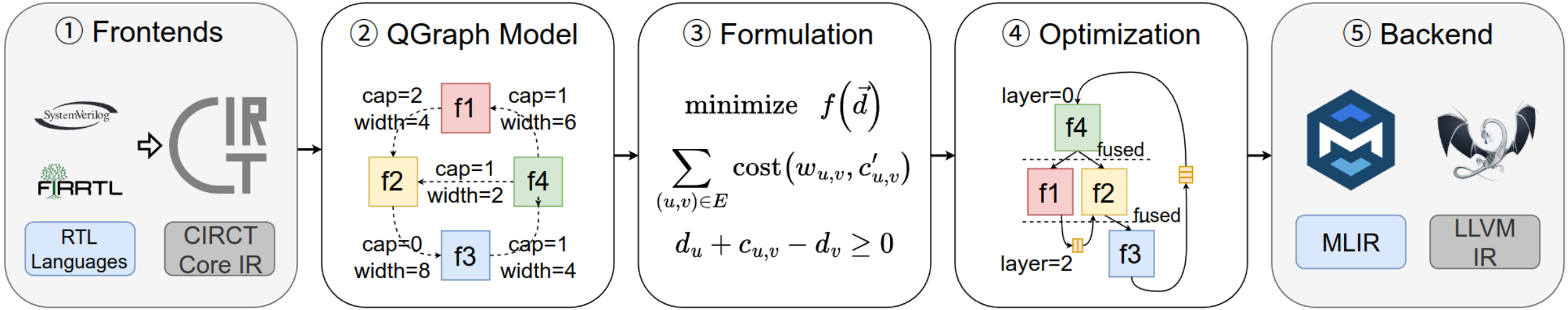


Fusing R/W by Changing Simulation Order

- In RTL Sim, state R/W is at cycle begin & end

- Adjusting Sim order, making R/W in the same iteration
  - data passed in reg, not memory

Overview of Khronos



Input: CIRCT Core IR

- Reusing frontends
- Fully support FIRRTL
- Partial support V/SV

Modeling & Formulation

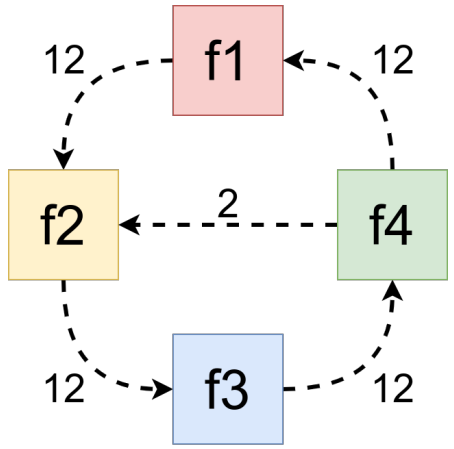
- Model RTL a depedency graph
- Log cost function for regbuffer

Optimization

- Linear Cons Non-linear Obj
- Iterative linearize for LP solver

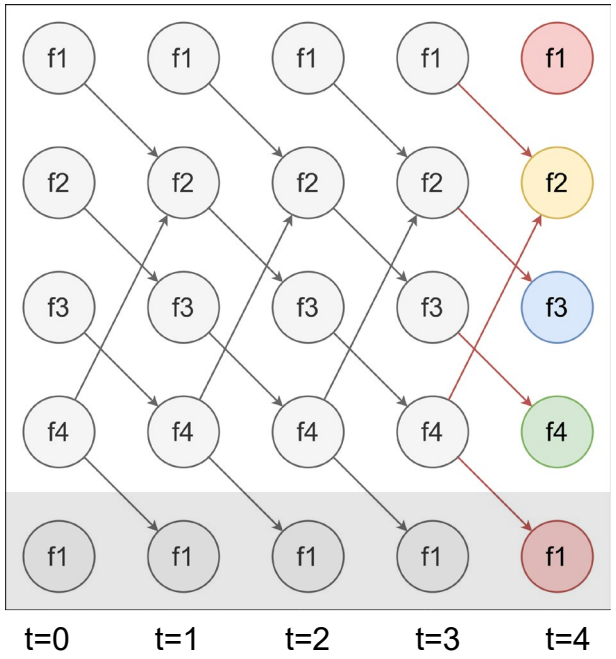
Output: LLVMIR

Modeling



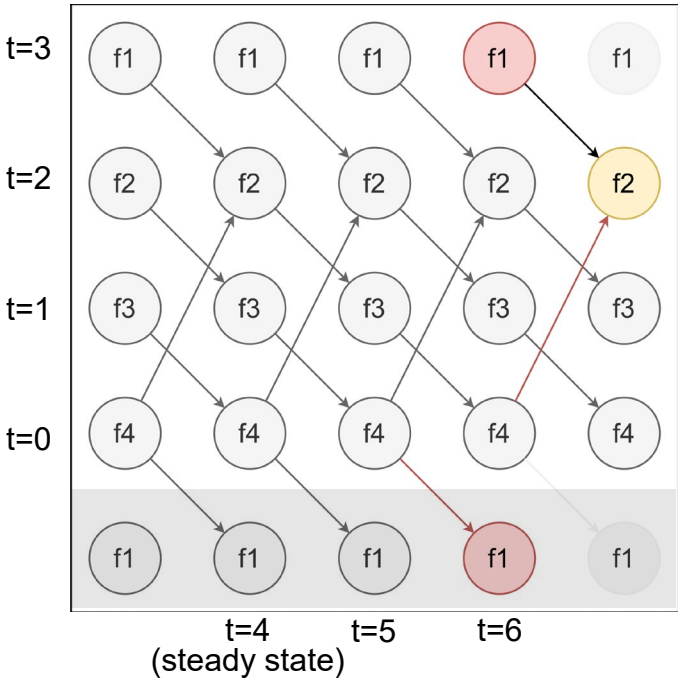
Example Pipelined Circuit

- f1, f2, f3, f4 are pipeline stages
- 12 word register between stages
- f4 forward 2 word to f2



Full Cycle Simulation

- Simulate all stage each iter
- *12x4+2=*50 word R/W each iter



Fused Full Cycle Simulation

- simulation some stage in advance
- only 12+2=14 word R/W

Optimization Algorithm

- Problem Formulation:

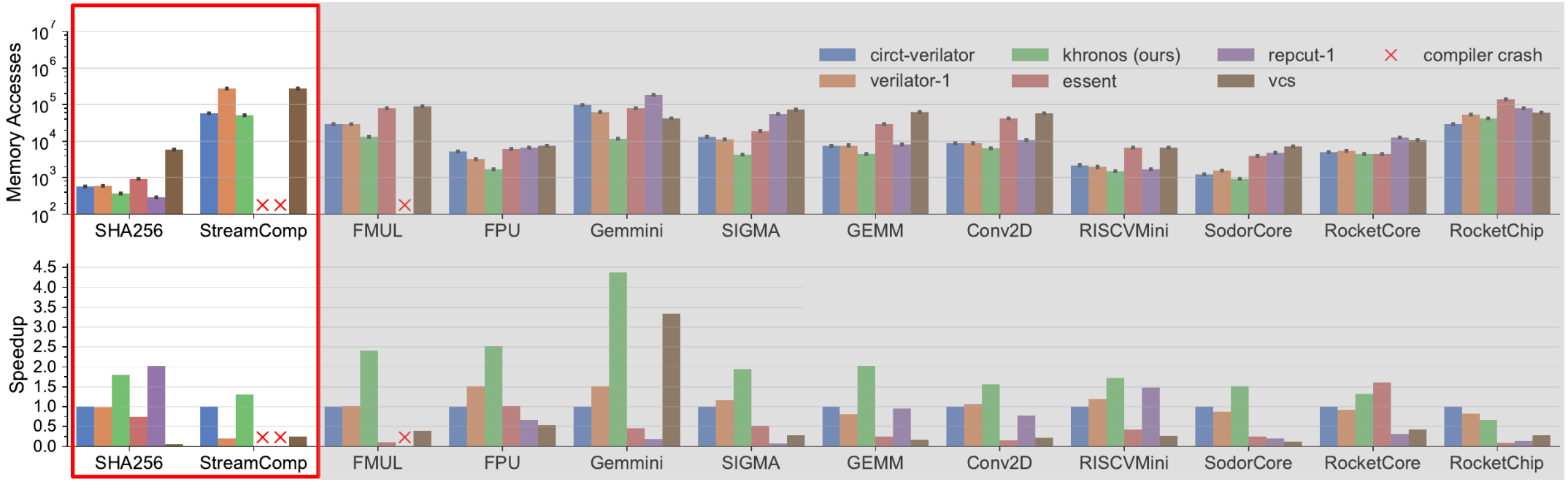
$$minimize f(\boldsymbol{d}) = \sum cost(u, v)$$
$$s.t. d_u + c_{u,v} - d_v \geq 0$$

- Optimization Algorithm: Iterative Linearization
  - Start at init guess , improve it each round
  - Cost linearization:
  - Run LP to get the next solution

$$minimize f'(\boldsymbol{d_i}) \cdot (\boldsymbol{d})$$
$$s.t. d_u + c_{u,v} - d_v \geq 0$$

  - Integer solution guarantee: unimodular

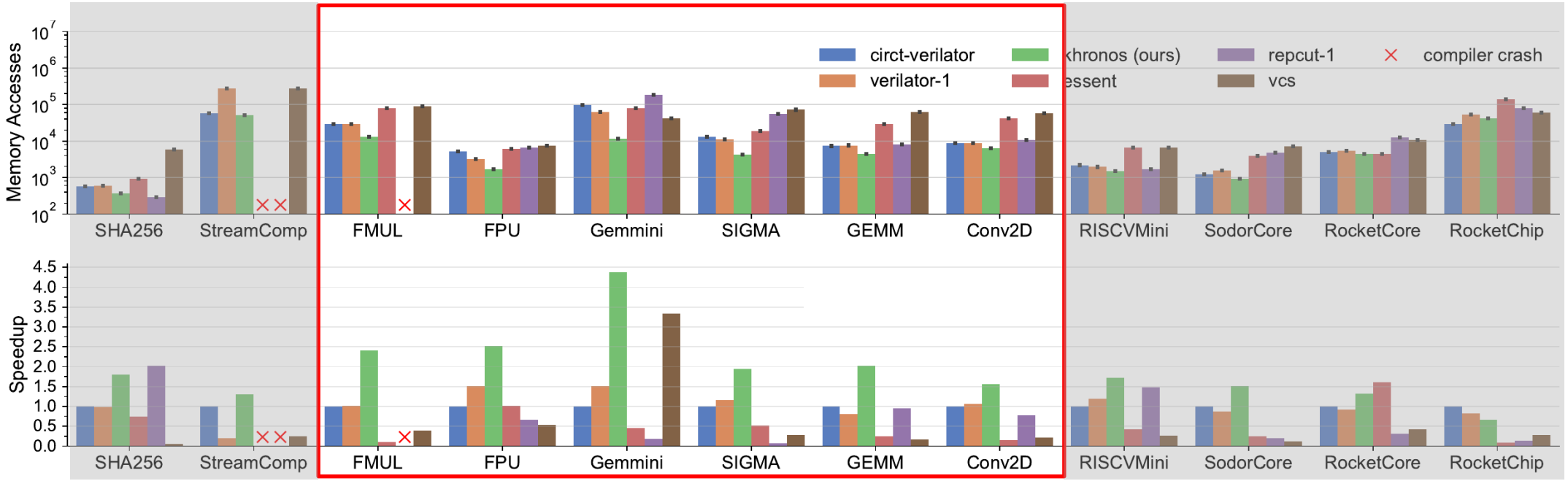
Experiment Results



Shallow pipeline

- ~20% fused state
- no enough state to be fused

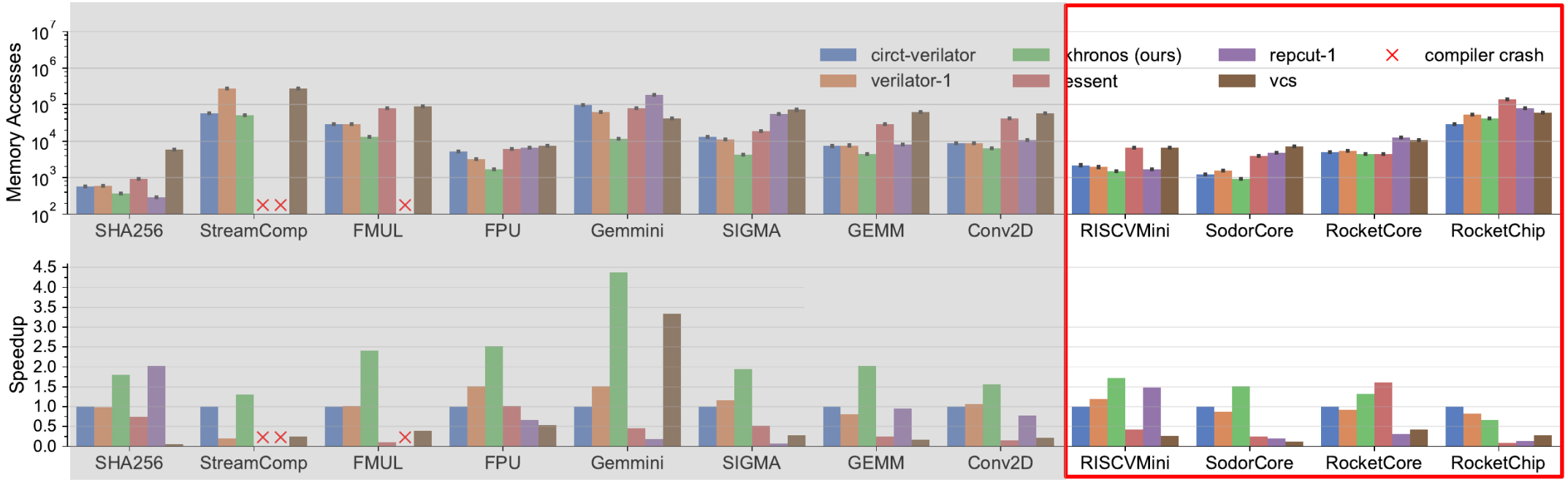
Experiment Results



Deep pipeline

- 40~80% fused state
- reduce 60~70% memory access
- 40~70% instruction reduction
- 1.5~4.3x acceleration

Experiment Results



Partily Pipelined

- 5~15% fused state
- reduce ~15% memory access
- 1.1~1.5x acceleration

AHS Resource

- Webpage: https://ericlyun.me/tutorial-aspdac2025/
  - Papers, presentation, code

- High-level Synthesis and DSL
  - ICCAD'22, FCCM'23, MICRO'24
- Hardware Simulation/Verification
  - MICRO'23
- Embedded Hardware Description Language
  - FPGA'24
- LLM-assisted RTL generation
  - ICCAD'24

Comparison

		Generality	Deterministic Timing	Control Logic Specification
Hardware Description Language (HDL)	(System)Verilog	*yes*	*no*	*manual*
	Chisel	*yes*	*no*	*manual*
	BSV(+Stmt)	*yes*	*no*	*procedural*
	Cement	***yes***	***yes***	***procedural***
	Filament	*limited*	*yes*	*timeline type*
High-level Synthesis (HLS)	HLS tools	*limited*	*no*	*software*
Domain-specific Language (DSL)	Dahlia	*limited*	*no*	*software*
	Spatial	*limited*	*no*	*software*
	Aetherling	*limited*	*yes*	*space-time type*
Hardware Intermediate Representation (IR)	Calyx	*limited*	*partial*	*procedural*

Definition

Deterministic timing indicates that the description deterministically dictates the occurrence of hardware operations during each cycle.

Productivity for control logic description

low median high

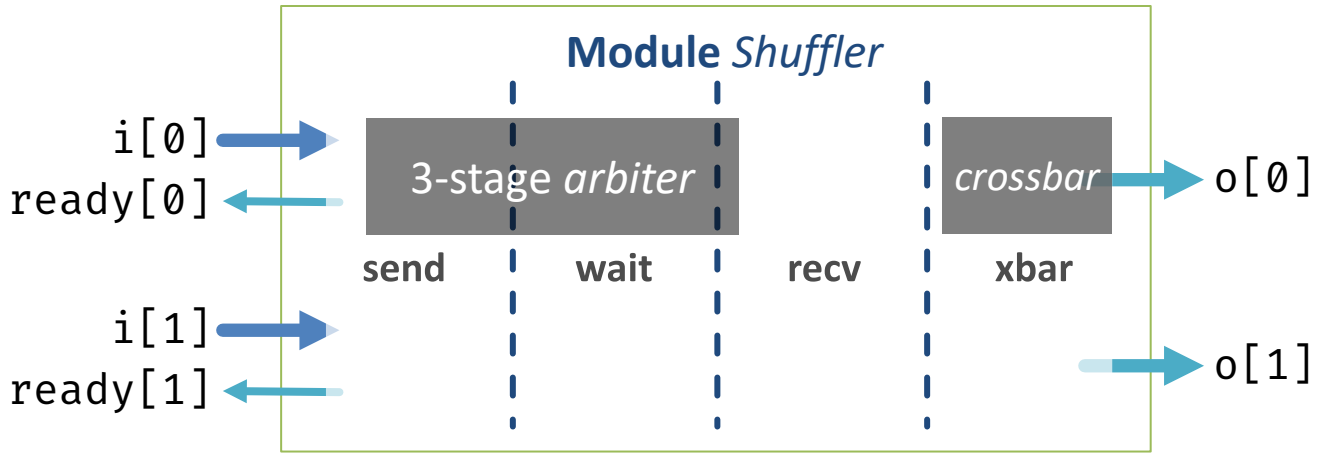
Shuffler in HDL

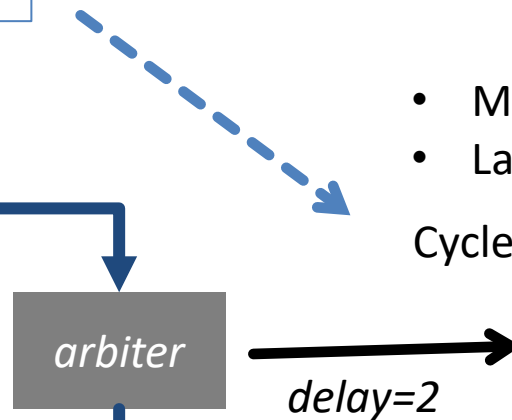
```
// Chisel
class Shuffler extends Module {
  val io=IO()
  val arbiter = Module()
  val crossbar = Module()

  val state = RegInit(0.U(4.W))
  state := Cat(state(2,0), io.go)

  val send = state(0)
  when (send) { // send stage
    resend(arb_in, io.ready,
         io.i, arb_out)
    arbiter.io.i := arb_i
  }

  val recv = state(3)
  when (recv) { // recv stage
    arb_o := arbiter.io.o
  }
  // other stages
}
```





- Manual control logic description as *FSM*
- Lack *timing* information

Cycle .. k k+1 k+2 k+3 ..

send recv recv recv

compiler cannot check!

Not specified at the language level!

Advantage
good generality

Disadvantage
not productive
error-prone

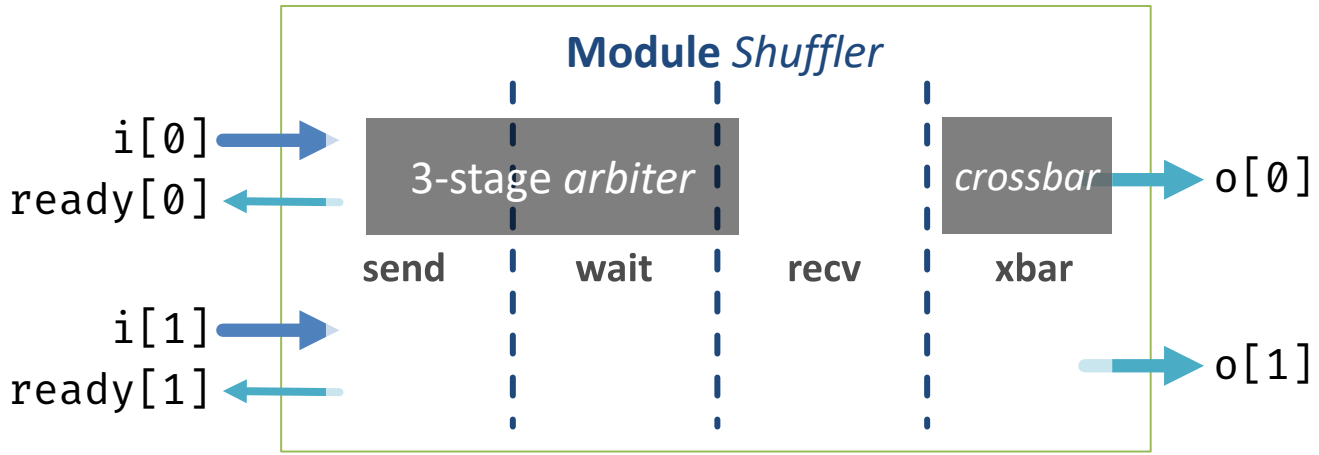
Shuffler in HLS

```
// C++
void shuffler(
  stream<Pkt>[N] &i,
  stream<Bit>[N] &ready,
  stream<Pkt>[N] &o) {
  while (!loop_exit) {
#pragma HLS pipeline II=1

    arbiter(arb_i, arb_o);

    for (int j = 0; j < N; j++) {
#pragma HLS unroll
    // resend logic:
    // arb_i and ready depends
    // on i and arb_o
    }

    crossbar(arb_o, o);
    // other logic
  }
}
```



mismatch !

- Synthesize *software* into control logic
- Users cannot determine *timing*

II=3

Cycle	..	k	k+1	k+2	k+3	..
		send	wait	recv	xbar	
				send	wait	recv	xbar

Advantage
productive

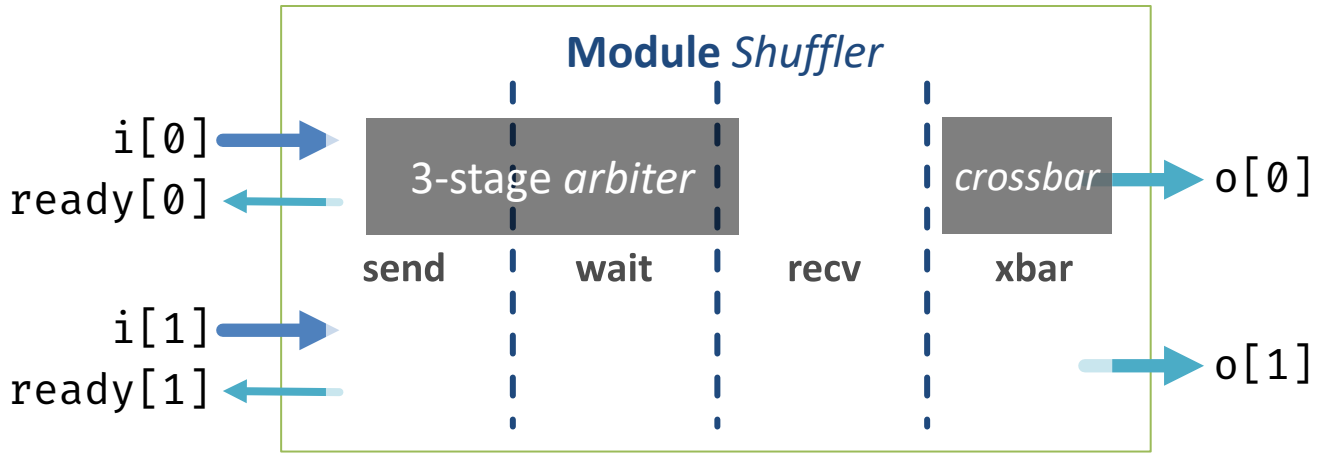
Disadvantage
limited generality
unpredictable

Shuffler in Cement

```
// CMTHDL
module! { IO =>
  shuffler(io) {
    let arbiter = instance!();

    let send = event! {
      // resend logic
      arbiter.i %= arb_i;
    };
    let recv = event! {
      arb_o %= arbiter.o;
    };
    // other stages

    let pipeline = stmt! {
      seq { {send} {wait} {recv} {xbar} }
    };
    synth!(pipeline,
      Pipeline::new(io.clk, io.go, II=1));
  }
}
```



- Event-based *procedural* control logic description
- Users specify *timing* deterministically

Cycle	..	k	k+1	k+2	k+3	..
		send	wait	recv	xbar	
		II=1	send	wait	recv	xbar

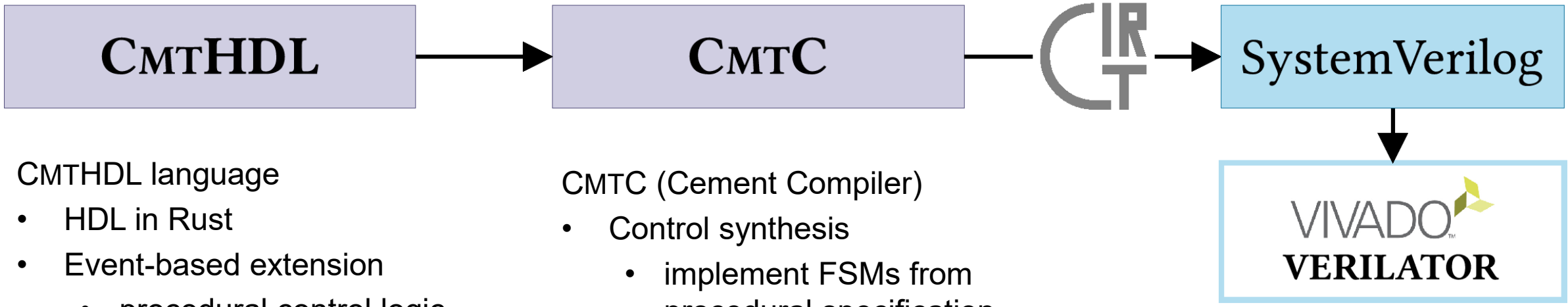
match !

Advantage

good generality deterministic timing

productive

Overview of Cement



CMTHDL language

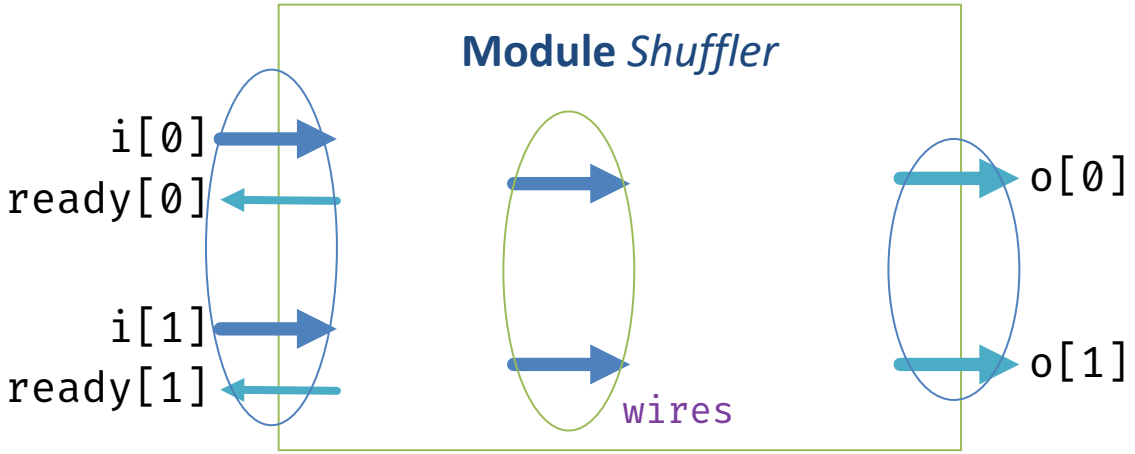
- HDL in Rust
- Event-based extension
  - procedural control logic specification
  - deterministic timing

CMTC (Cement Compiler)

- Control synthesis
  - implement FSMs from procedural specification

Ports/Wires

- Ports and Wires are Rust types

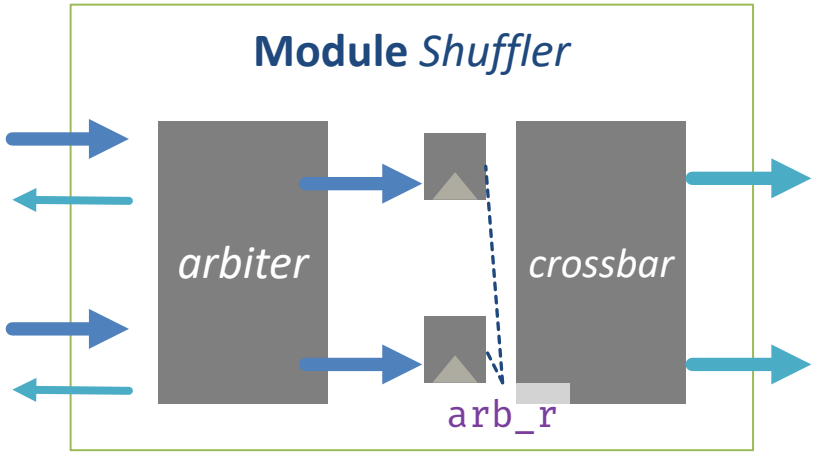



```
#[interface(Default)]
pub struct IO<const N: usize, T: DataType> {
  clk: Clk,
  i: [Pkt<N,T>; N], // in
  ready: <[B<1>; N] as Interface>::FlipT, // out
  o: <[Pkt<N,T>; N] as Interface>::FlipT, // out
}
```

Hardware: module ports/wires

Submodule and Wire Connection

- Instantiation




```
// CMTHDL
module! { IO =>
  shuffler(io) {
    let arbiter = instance!(arbiter(ArbIO::new()));
    let crossbar = instance!(crossbar(XbarIO::new()));

    let arb_r = reg!(arbiter.o.ifc(), io.clk);

    let recv = event! {
      arb_r.wr %= arbiter.o;
    };

    // other stages and control logic
}}
```

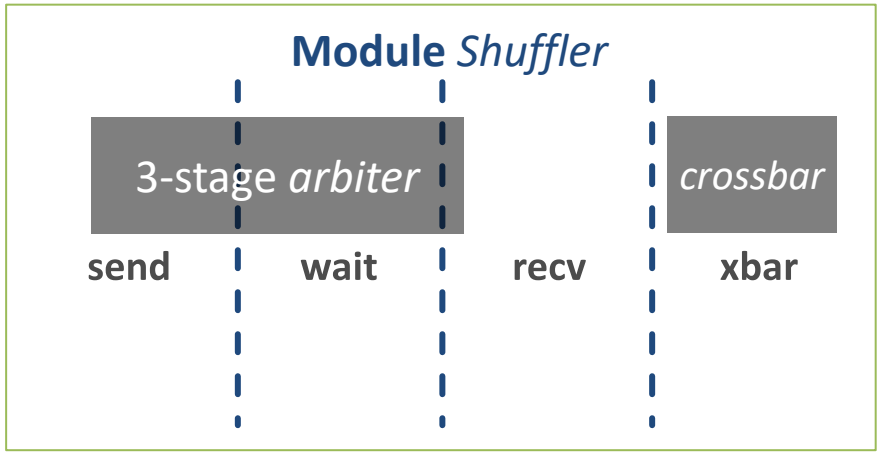
instantiate submodule

instantiate register

specify connection

Procedural Control Logic Specification

- Ctrl sub-language to specify control logic as event-based procedural statements




```
shuffler(io) {
  let send = event! {};
  let wait = event! {};
  let recv = event! {};
  let xbar = event! {};
  let pipeline =
     stmt! {
       seq {{send} {wait} {recv} {xbar}}
     };
  synth!(pipeline,
       Pipeline::new(io.clk, io.go, II=1));
}
```

An ***event*** is a group of hardware operations that occur simultaneously.

Pipeline is specified as ***a "sequence" of 4 steps*** in the ***ctrl*** sub-language (**`stmt!`**)

Deterministic Timing

Statements	step	seq	par	if	for	while

Macro syntax	(e_entry)
	e0, e1, ..
	(e_exit)
		seq {
		{s0}
		{s1}
		..
		}
			par {
			{s0}
			{s1}
			..
			}
				if cond =>
				t_stmt
				else
				e_stmt
					for indvar
					in range =>
					do_stmt
						while cond =>
						do_stmt

Semantic	Wait until e_entry happens, trigger e0,e1, .. **in one cycle**, then wait until e_exit.	Trigger s0,s1, .. **sequentially without interval**.	Trigger s0,s1, .. **immediately**, wait until all of them finishes.	Trigger t_stmt or e_stmt **immediately** if cond happens or not.	**Repeat** do_stmt **without interval** according to range.	**Repeat** do_stmt **without interval** until cond fails.

```
seq {
  {step0}
  {step1}
  {step2}
}
```

Cycle	..	k	k+1	k+2	k+3	..
		step0	step1	step2		

Definition

Deterministic timing indicates that the description deterministically dictates the occurrence of hardware operations during each cycle.

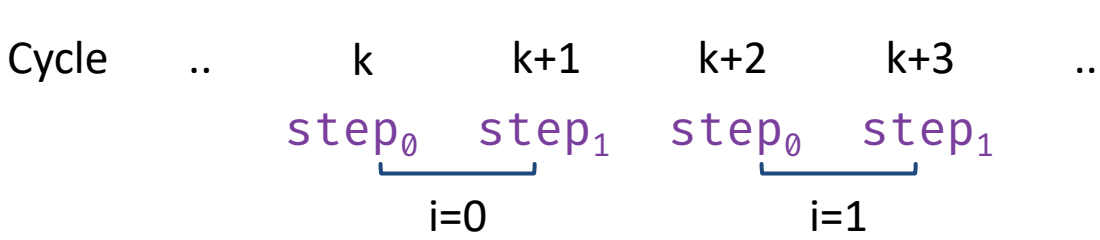
Deterministic Timing

Statements	step	seq	par	if	for	while

Macro syntax	(e_entry)
	e0, e1, ..
	(e_exit)
		seq {
		{s0}
		{s1}
		..
		}
			par {
			{s0}
			{s1}
			..
			}
				if cond =>
				t_stmt
				else
				e_stmt
					for indvar
					in range =>
					do_stmt
						while cond =>
						do_stmt

Semantic	Wait until e_entry happens, trigger e0,e1, .. **in one cycle**, then wait until e_exit.	Trigger s0,s1, .. **sequentially without interval**.	Trigger s0,s1, .. **immediately**, wait until all of them finishes.	Trigger t_stmt or e_stmt **immediately** if cond happens or not.	**Repeat** do_stmt **without interval** according to range.	**Repeat** do_stmt **without interval** until cond fails.

```
for i in 0..2 =>
  seq {
    {step0}
    {step1}
  }
```

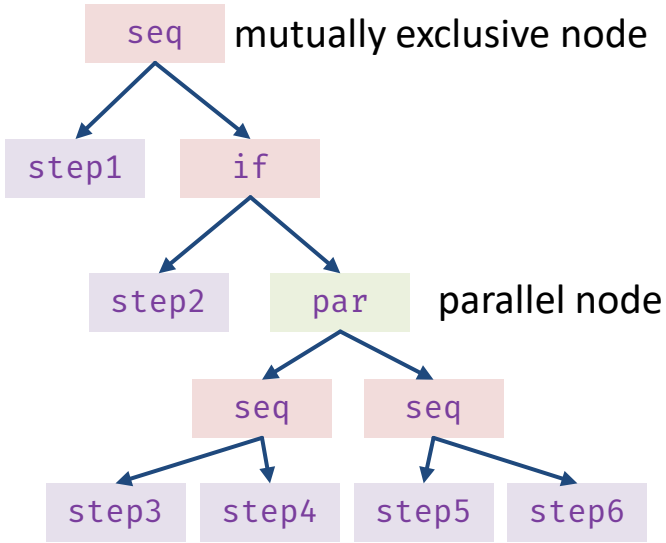


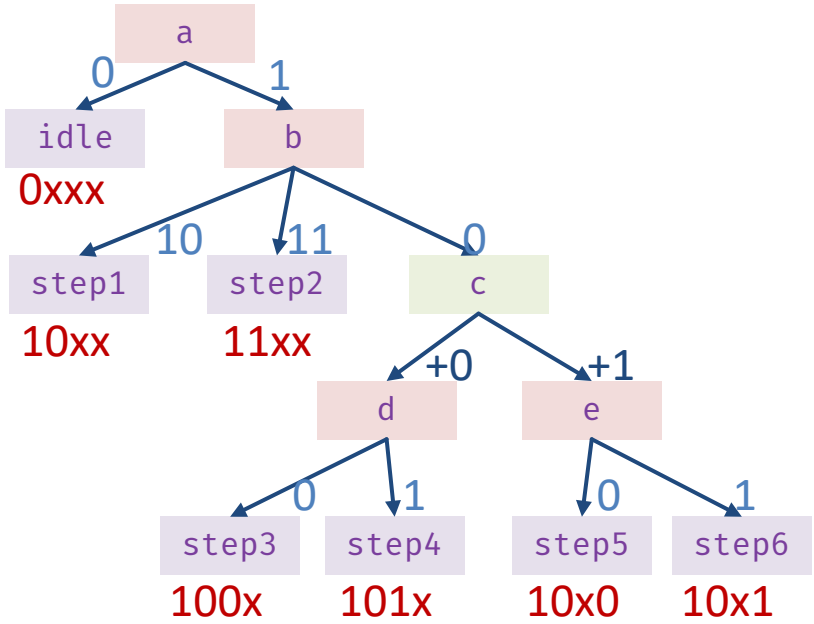
Definition

Deterministic timing indicates that the description deterministically dictates the occurrence of hardware operations during each cycle.

Control Synthesis

```
let s = stmt! {
  seq {
    {step1}
    {if cond =>
        step2
      else
        par {
          {seq {
            {step3}
            {step4}
          }}
          {seq {
            {step5}
            {step6}
          }}
        }
    }
  }
};
synth!(s, omitted);
```



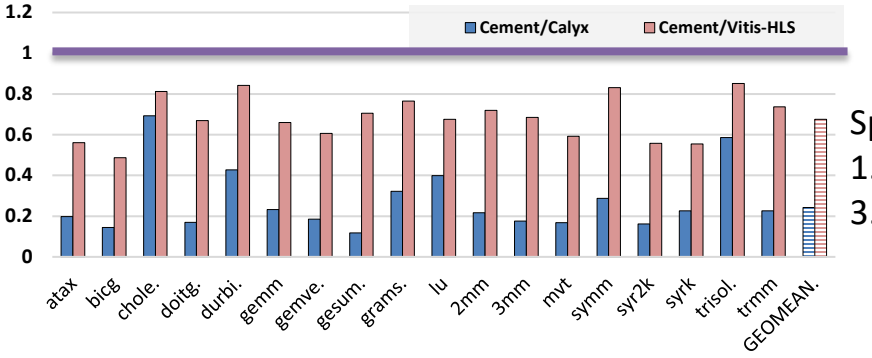


Procedural description → AST → State tree

FSM optimized for *frequency* and *resource efficiency* on FPGAs

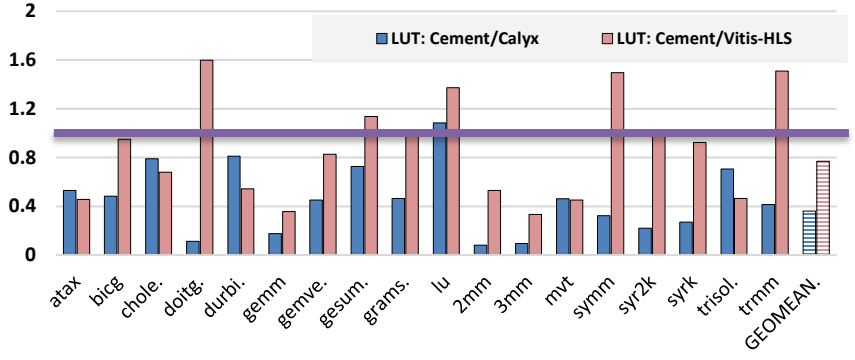
Evaluation: PolyBench Results

Performance (cycle count)



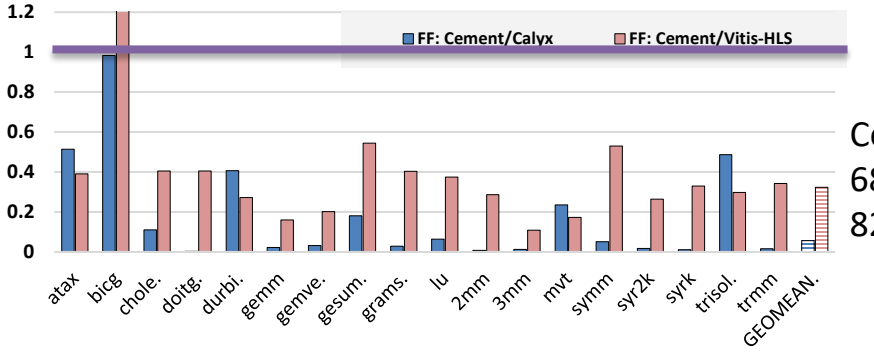
Speedup
1.41× vs. Vitis-HLS
3.49× vs. Dahlia-Calyx

Resource (LUT)



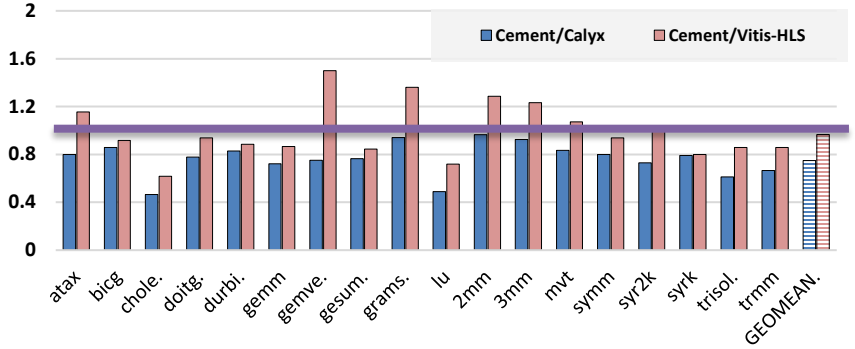
Cement saves LUT
23% vs. Vitis-HLS
54% vs. Dahlia-Calyx

Resource (FF)



Cement saves FF
68% vs. Vitis-HLS
82% vs. Dahlia-Calyx

Productivity (lines of code)

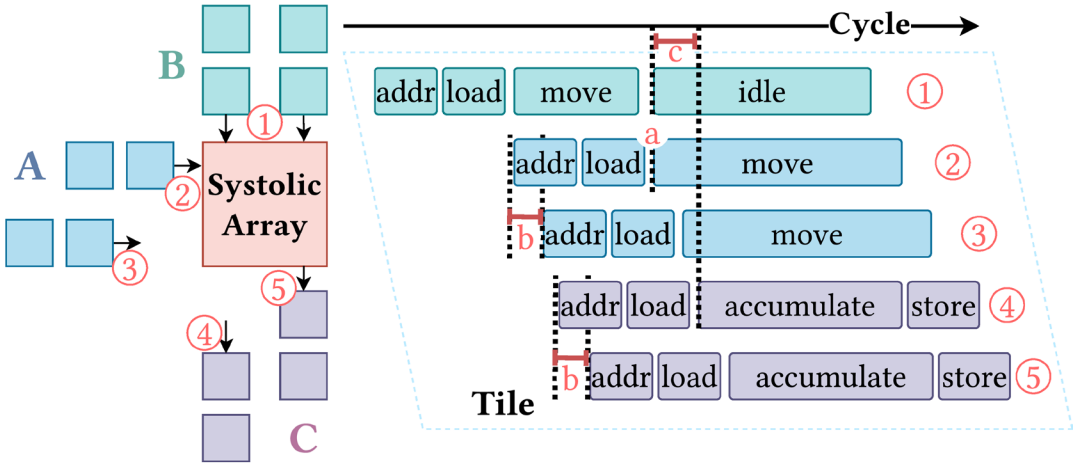


Cement saves LoC
3% vs. Vitis-HLS
25% vs. Dahlia-Calyx

The y-axis represents the ratio of *Cement* and the other two methods (***Dahlia-Calyx flow*** and ***Vitis HLS***).
A **smaller** value *(<1)* means that *Cement* has **better** results.

Evaluation: Case Study on Systolic Array

Cement helps to specify timing relations (a , b , c)



Design	LUT	DSP	Frequency	Throughput
AutoSA (FPGA '21)	968k	9462	272MHz	949.98 GFLOPS
EMS (DAC '22)	898k	4494	301MHz	731.17 GFLOPS
$\text{Cement}_{\text{small}}$	437k	3840	322MHz	823.97 GFLOPS
$\text{Cement}_{\text{large}}$	543k	4800	333MHz	1065.60 GFLOPS

Fewer resource

$\text{Cement}_{\text{small}}$ vs. EMS-WS: 51% ↓ LUTs, 15% ↓ DSPs
$\text{Cement}_{\text{large}}$ vs. AutoSA: 44% ↓ LUTs, 49% ↓ DSPs

Better performance

$\text{Cement}_{\text{small}}$ vs. EMS-WS: 7% ↑ frequency, 13% ↑ throughput
$\text{Cement}_{\text{large}}$ vs. AutoSA: 22% ↑ frequency, 12% ↑ throughput

Better productivity

(Cement) 2 person-month vs. (EMS) 6 person-month

AHS Resource

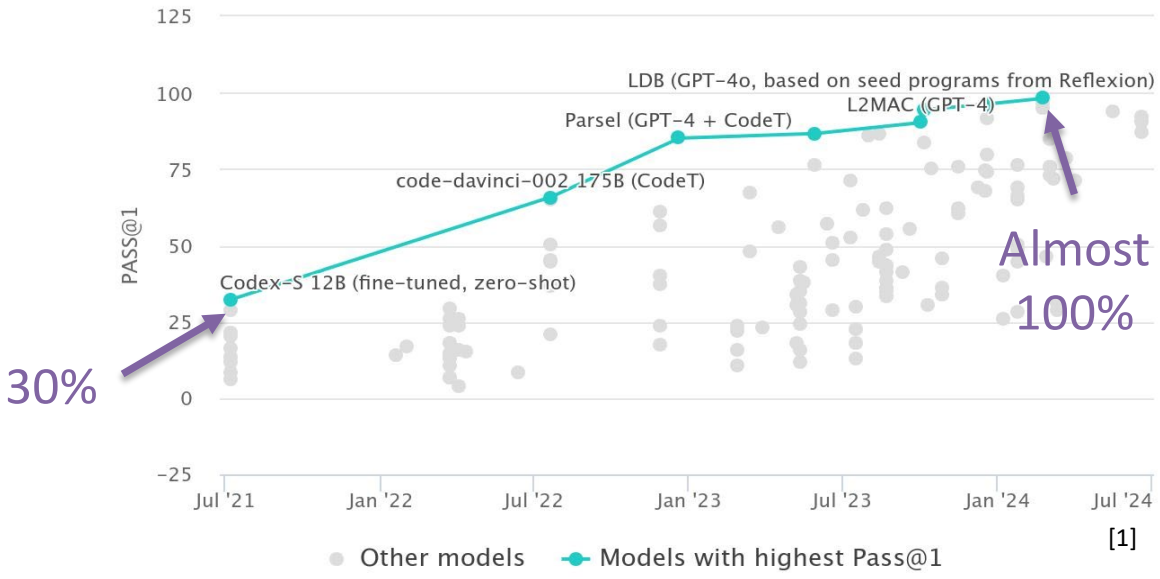
- Webpage: https://ericlyun.me/tutorial-aspdac2025/
  - Papers, presentation, code

- High-level Synthesis and DSL
  - ICCAD'22, FCCM'23, MICRO'24
- Hardware Simulation/Verification
  - MICRO'23
- Embedded Hardware Description Language
  - FPGA'24
- LLM-assisted RTL generation
  - ICCAD'24

LLM-Driven Code Generation

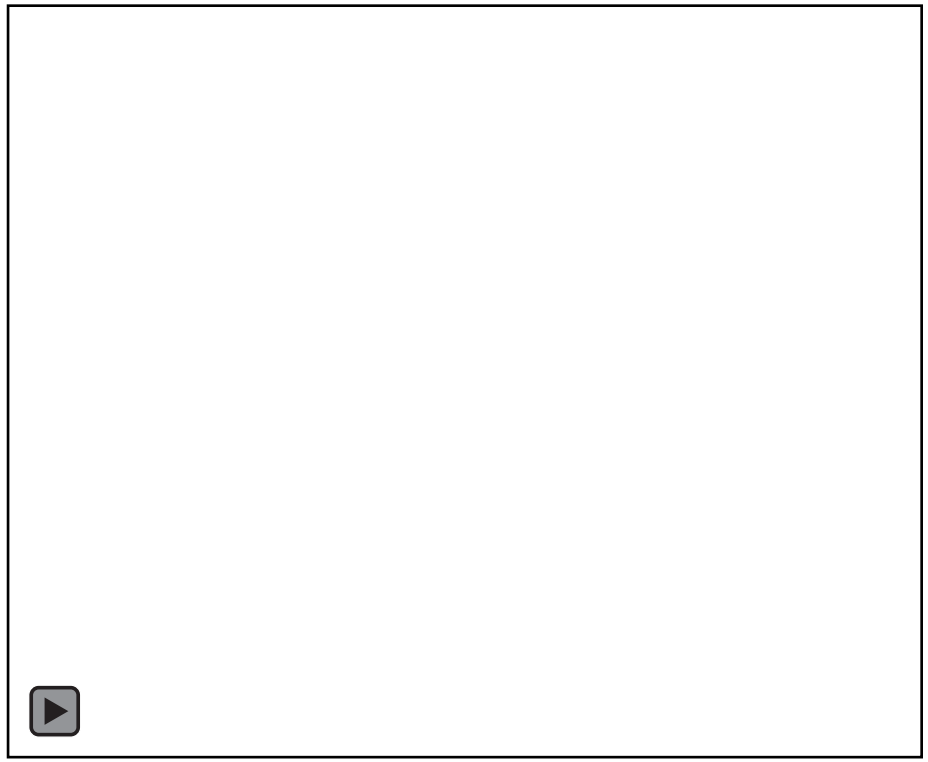
- The coding capabilities of LLM have significantly improved
- LLM-powered coding assistant transform modern software development

Code Generation on HumanEval(Python)



[1]

GitHub Codepilot

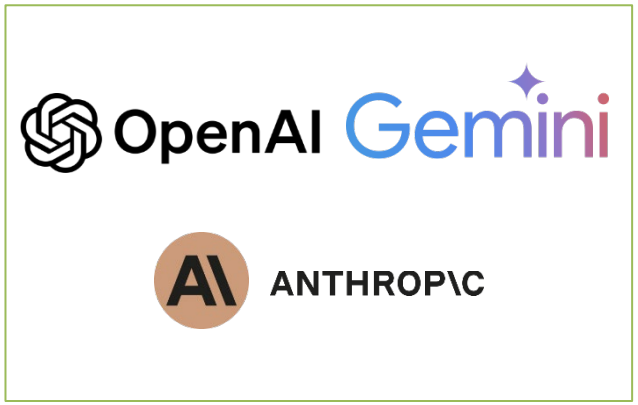


Open-Source Models vs Closed-Source Models

Open-Source Models

- Full Control and Transparency
- Customizability
- Almost Free
- Low Performance

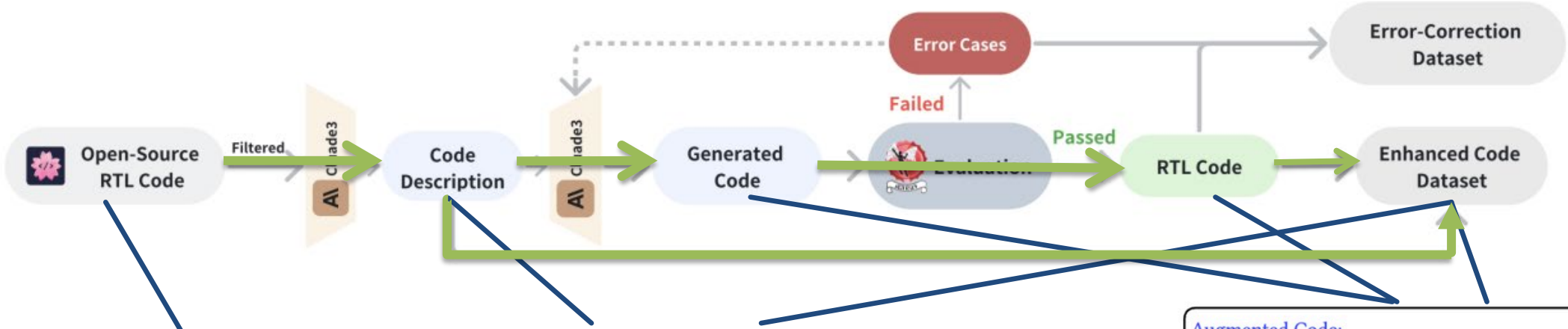
Closed-Source Models



- Privacy and security concerns
- Lack of Customizability
- Costly
- High Performance

Bridging the performance gap between open-source and closed-source models

Overview of Origen



Original Code:

```
module flip_flop(clk,d,q, rst, ena);
input  wire clk, rst, ena;
input  wire d;
output reg  q;
always @(posedge clk) begin
    if(rst) begin
        q <= 0;
    end
    else begin
        if(ena) begin
            q <= d;
        end
    end
end
endmodule
```

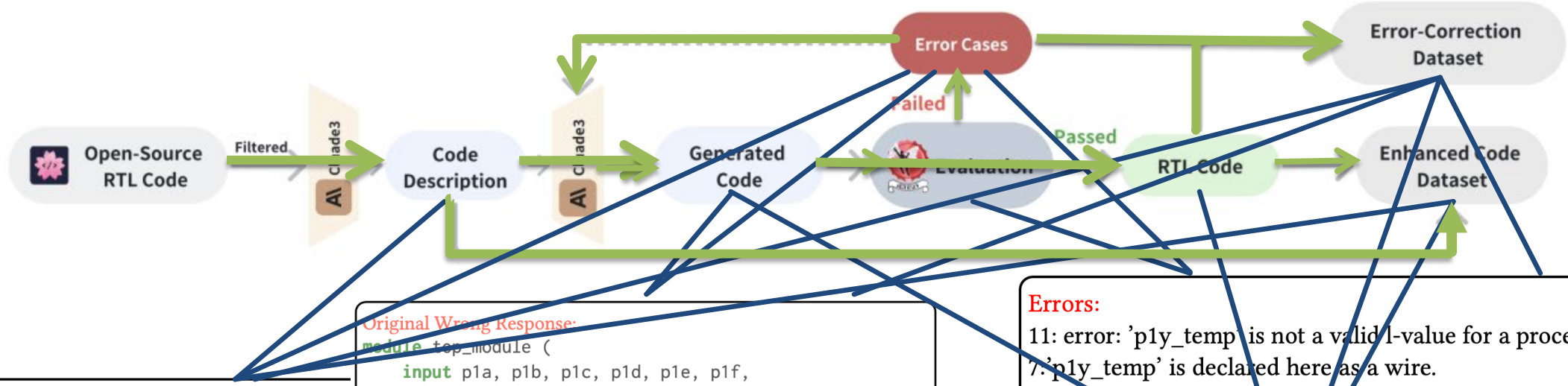
Description (Generated by LLM):

This Verilog module is a flip-flop with synchronous reset and enable signals. It updates its output 'q' based on the input 'd' on the rising edge of the clock signal 'clk'. If the reset signal 'rst' is asserted, the output 'q' is set to 0. Otherwise, if the enable signal 'ena' is asserted, the output 'q' is updated with the value of the input 'd'.

Augmented Code:

```
module flip_flop (
    input  wire clk,
    input  wire d,
    input  wire rst,
    input  wire ena,
    output reg  q
);
    always @(posedge clk) begin
        if (rst) begin
            q <= 1'b0;
        end
        else if (ena) begin
            q <= d;
        end
    end
endmodule
```

Overview of Origen



Task Name: 7458

The 7458 is a chip with four AND gat
module in Verilog with the same funct
10 inputs and 2 outputs. In this circuit
3-input AND gates: one that ANDs p1
that ANDs p1d, p1e, and p1f. The outp
AND gates: one that ANDs p2a and p2b
and p2d.

Original Wrong Response:

```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y
);
    wire p1y_temp;
    assign p1y = p1y_temp;
    always @(p1a or p1b or p1c or p1d or p1e or p1f)
    begin
        p1y_temp = (p1a&p1b&p1c) | (p1d&p1e&p1f);
    end
    wire p2y_temp;
    assign p2y = p2y_temp;
    always @(p2a or p2b or p2c or p2d)
    begin
        p2y_temp = (p2a & p2b) | (p2c & p2d);
    end
endmodule
```

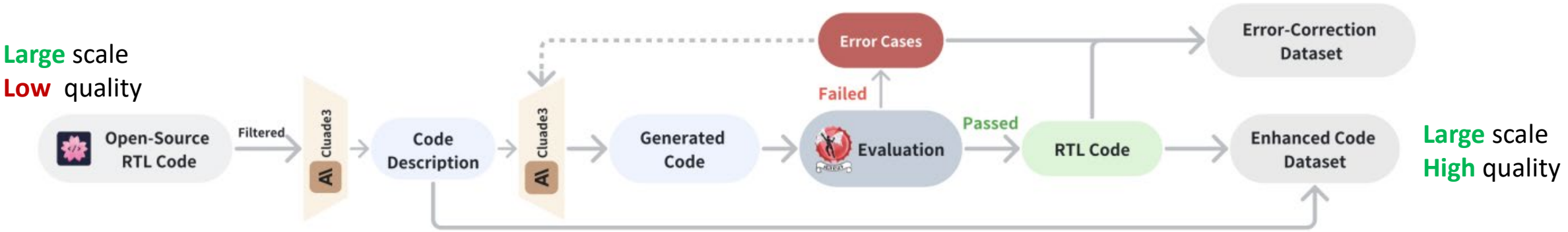
Errors:

11: error: 'p1y_temp' is not a valid l-value for a procedural assignment.
7:'p1y_temp' is declared here as a wire.
17: error: 'p2y_temp' is not a valid l-value for a procedural assignment.
13:'p2y_temp' is declared here as a wire.

Regenerated Code:

```
module top_module (
    input p1a, p1b, p1c, p1d, p1e, p1f,
    output p1y,
    input p2a, p2b, p2c, p2d,
    output p2y
);
    wire p1y_temp;
    assign p1y = (p1a&p1b&p1c) | (p1d&p1e&p1f);
    wire p2y_temp;
    assign p2y = (p2a & p2b) | (p2c & p2d);
endmodule
```

Overview of Origen



- The augmentation process enhances the quality
  - Harness the capabilities of closed-source LLM
- Filter the generated code to further enhance its quality
- Two Datasets
  - One for RTL generation
  - One for RTL syntax error fix

Evaluation: VerilogEval and RTLLM

- Significantly outperform other Verilog-specific models
- Outperform the model Claude3-haiku used for synthesizing data
- Slightly inferior to the GPT-4 Turbo and Claude3-Opus

Table 1: Comparison of functional correctness on VerilogEval [13] and RTLLM [16]

Source	Name	VerilogEval-human(%)			VerilogEval-machine(%)			RTLLM(%)
		pass@1	pass@5	pass@10	pass@1	pass@5	pass@10	pass@5
Commercial LLM	GPT-3.5 [1]	35.6	48.8	52.6	49.4	72.7	77.6	44.8
	GPT-4 2023-06-13 [1]	43.5	55.8	58.9	60.0	70.6	73.5	65.5
	GPT-4 Turbo 2024-04-09 [1]	54.2	**68.5**	**72.4**	58.6	71.9	76.2	65.5
	Claude3-Haiku [2]	47.5	57.7	60.9	61.5	75.6	79.7	62.1
	Claude3-Sonnet [2]	46.1	56.0	60.3	58.4	71.8	74.8	58.6
	Claude3-Opus [2]	**54.7**	63.9	67.3	60.2	75.5	79.7	**69.0**
Open Source Models	CodeLlama-7B-Instruct [20]	18.2	22.7	24.3	43.1	47.1	47.7	34.5
	CodeQwen1.5-7B-Chat [3]	22.4	41.1	46.2	45.1	70.2	77.6	37.9
	DeepSeek-Coder-7B-Instruct-v1.5 [9]	31.7	42.8	46.8	55.7	73.9	77.6	37.9
Verilog-Specific Models	ChipNeMo [12]	22.4	-	-	43.4	-	-	-
	VerilogEval [13]	28.8	45.9	52.3	46.2	67.3	73.7	-
	RTLCoder-DeepSeek [14]	41.6	50.1	53.4	61.2	76.5	81.8	48.3
	CodeGen-6B MEV-LLM [17]	42.9	48.0	54.4	57.3	61.5	66.4	-
	BetterV-CodeQwen [19]	46.1	53.7	58.2	68.1	79.4	84.5	-
OriGen (ours)		51.4	58.6	62.2	**76.2**	**84.0**	**86.7**	65.5
OriGen (updated)		54.4	60.1	64.2	**74.1**	**82.4**	**85.7**	69.0

Schedule

Time	Agenda	Presenter
50mins	Overview of AHS	Yun Liang
	Hands-on Session	
45mins	High-level Synthesis (ICCAD'22, FCCM'23, MICRO'24)	Ruifan Xu and Xiaochen Hao
20mins	Hardware Simulation (MICRO'23)	Kexing Zhou
45mins	Hardware Description Language (FPGA'24)	Youwei Xiao and Zizhang Luo
20mins	LLM-based Chip Design (ICCAD'24)	Fan Cui

Setup

- A: Download the docker image
  - Please follow the handout or our website
  - https://ericlyun.me/tutorial-aspdac2025
- B: Login our server
  - https://ahs.ericlyun.me
  - User: ahs-aspdac25
  - Pass: AgileChipDesign
  - Enter your name and ssh key
  - Follow the output to connect