Modeling and Simulation

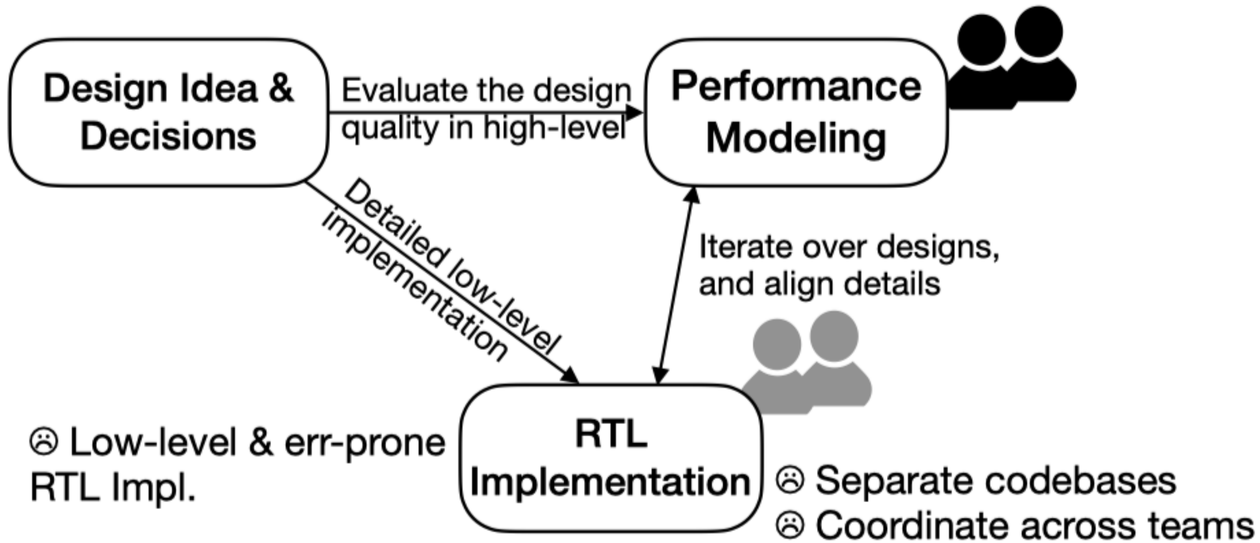
Weiming Hu

2025-07-25

Background

A Typical Architectural Design & Implementation Flow

- Design Idea & Decisions
  - Need to be evaluated in high-level, by architect
- Performance Modeling
  - Iterate over designs, and align details, by architect
- RTL Implementation
  - Iterate over designs, and align details by Design Engineer and Design Verification



Background

Chip Design Flow

- Simulation-based Modeling
  - Pipeline Organization
  - Design Parameters
  - Capture some cycle-level insights

- RTL Implementation
  - Is the design real?
  - Implement the Pipeline
  - Fronted verification
  - Simulating hardware concurrency

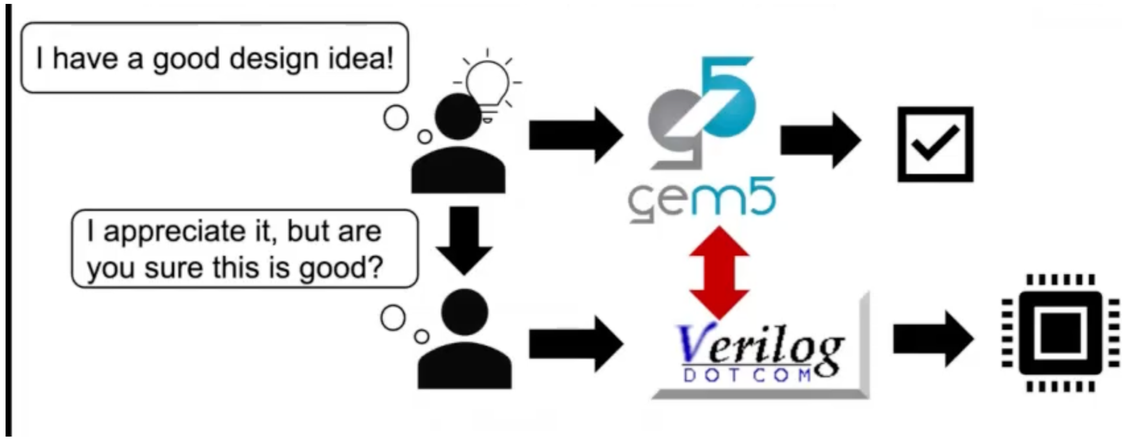
- Synthesis, Timing, and Verification
  - Synthesize into netlist
  - Gate-level verification
  - Does timing close?
  - Capture lowest level details

Source: https://b23.tv/UpyKD92

Background

A Typical Architectural Design & Implementation Flow

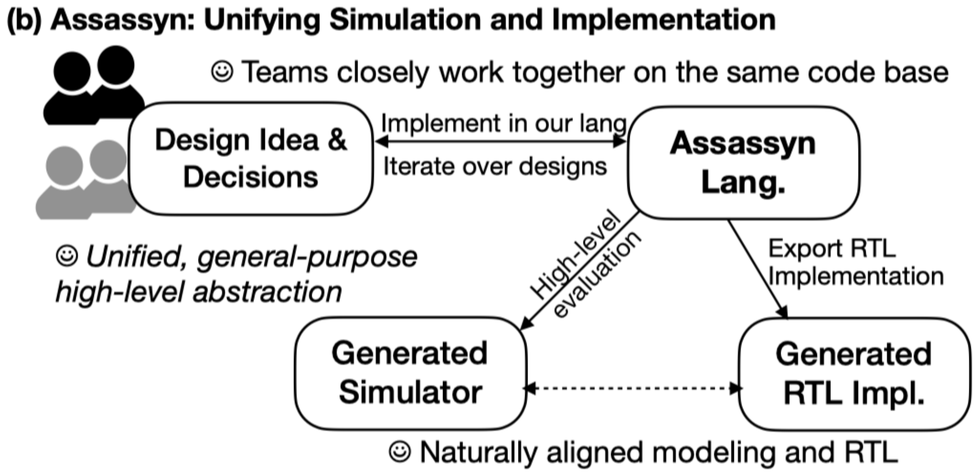
- Architecture research paper
  - Reporting performance numbers from simulation-based modeling
    - implementing in-house
    - extending existing open-source simulator
  - Reporting the power/area evaluations from separate synthesized RTL implementation
- Motivation
  - the alignment between the simulation and hardware implementation is often overlooked, highlighting a persistent challenge of this two-pronged approach.
  - **A gap between simulation and hw implementation**



Assassyn, ISCA 2025

A Typical Architectural Design & Implementation Flow

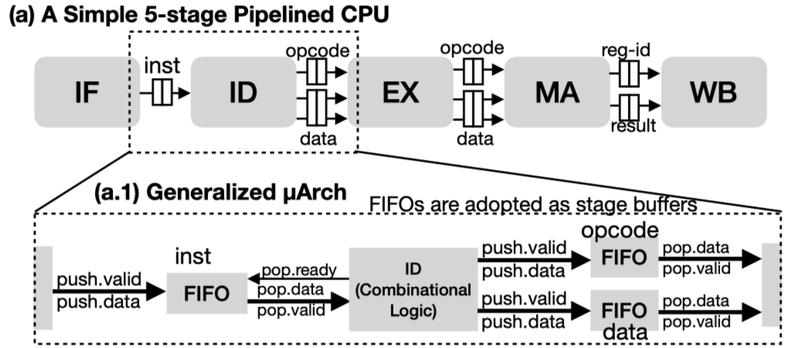
- Goal
  - an ideal architectural design and implementation workflow shall have a unified, general-purpose, and high-level language to describe the architecture design to generate both cycle-accurate simulation and RTL implementation
  - Teams closely work together on the same code base
  - Unified, general-purpose high-level abstraction
  - Naturally aligned modeling and RTL
- Prior work
  - they all failed to be general-purpose,



Motivation

A Typical Architectural Design & Implementation Flow

- Event Driven Modeling
  - Engine: Event queue; while loop
  - Cycle 1: event = q.pop() -> IF.simulate(q, event)-> pc +=4, inst=mem[pc]-> **q.push(Event(ID, inst))**
  - Cycle 2: event = q.pop() -> ID.simulate(q, event)-> **q.push(Event(EX, data, opcode))**
  - **Push/pop is similar to asynchronously invoking a function**
- RTL Programming
  - event-listen paradigm
  - pull in data to process their state machine
- This pull/push mismatch creates a significant gap between architecture design and implementation



(b) Event-driven Modeling

```
class IF:
  def sim(q, event):
    pc += 4
    inst = mem[pc]
    q.push(Event(ID, inst))
```

```
class ID:
  def sim(q, event):
    # decode event.inst
    e = Event(EX,
      data, opcode)
    q.push(e)
```

```
def engine():
  # key by Event.stamp
  q = PriorityQueue()
  while !q.empty():
    event = q.pop()
    if event.kind == IF:
      IF.simulate(q, event)
    elif event.kind == ID:
      ID.simulate(q, event)
    elif ...:
```

(c) SystemVerilog (FIFO impl & valid/ready omitted)

```
module IF(output inst);
  always @(posedge clk)
    inst <= mem[pc]; /*...*/
endmodule
```

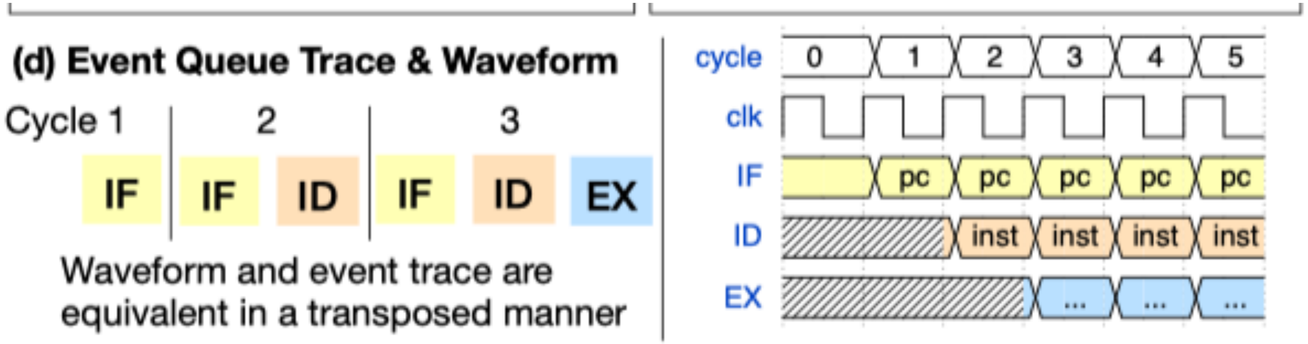
```
module ID(
 input inst,
 output opcode, output data,
); /*...*/
endmodule
```

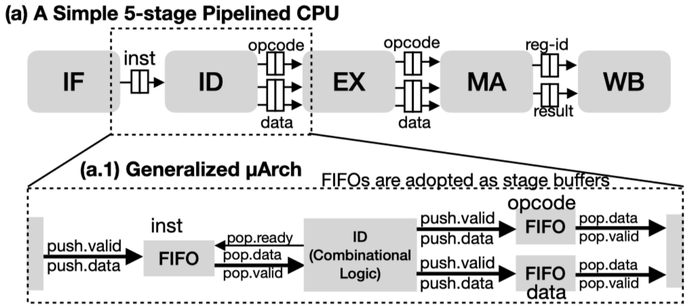
```
module top();
 logic inst_o;
 logic inst_i;
 IF fetcher(.inst(inst_o));
 StageFIFO if_id(
  .in(inst_o), .out(inst_i));
 ID decoder(
  .inst(inst_i),
  // Connect to EX stage
  .opcode(...), .data(...));
endmodule
```

Motivation

What is the same point?

- Trace **(output)** of event-drivent simulation **(input)**
- Waveform of**(output)** RTL programming **(input)**
- This correspondence presents an opportunity to create **a unified abstraction** to bridge the simulation and RTL implementation.





(b) Event-driven Modeling

```
class IF:
  def sim(q, event):
   pc += 4
   inst = mem[pc]
   q.push(Event(ID, inst))
```

```
class ID:
  def sim(q, event):
    # decode event.inst
    e = Event(EX,
      data, opcode)
    q.push(e)
```

```
def engine():
  # key by Event.stamp
  q = PriorityQueue()
  while !q.empty():
    event = q.pop()
    if event.kind == IF:
      IF.simulate(q, event)
    elif event.kind == ID:
      ID.simulate(q, event)
    elif ...:
```

(c) SystemVerilog (FIFO impl & valid/ready omitted)

```
module IF(output inst);
  always @(posedge clk)
    inst <= mem[pc]; /*...*/
endmodule
```

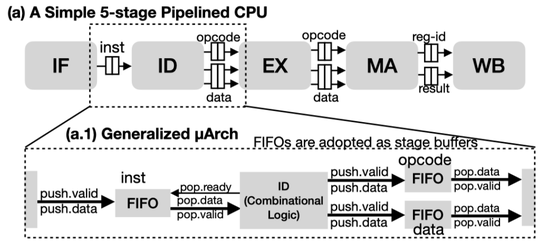
```
module ID(
 input inst,
 output opcode, output data,
); /*...*/
endmodule
```

```
module top();
 logic inst_o;
 logic inst_i;
 IF fetcher(.inst(inst_o));
 StageFIFO if_id(
  .in(inst_o), .out(inst_i));
 ID decoder(
  .inst(inst_i),
  // Connect to EX stage
  .opcode(...), .data(...));
endmodule
```

Motivation

Event-driven Programming

- Trace **(output)** of event-drivent simulation **(input)**
- Waveform of**(output)** RTL programming **(input)**
- This correspondence presents an opportunity to create **a unified abstraction** to bridge the simulation and RTL implementation.



(b) Event-driven Modeling

```
class IF:
  def sim(q, event):
   pc += 4
   inst = mem[pc]
   q.push(Event(ID, inst))
```

```
class ID:
  def sim(q, event):
    # decode event.inst
    e = Event(EX,
      data, opcode)
    q.push(e)
```

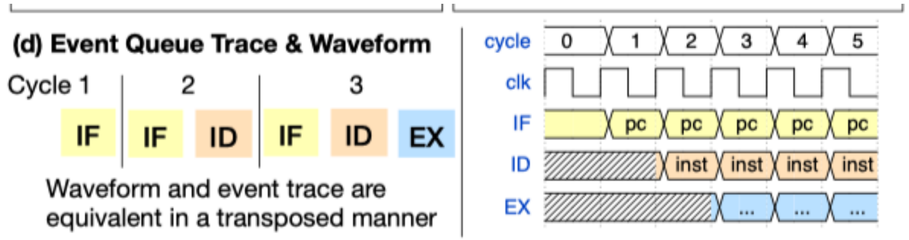
```
def engine():
  # key by Event.stamp
  q = PriorityQueue()
  while !q.empty():
    event = q.pop()
    if event.kind == IF:
      IF.simulate(q, event)
    elif event.kind == ID:
      ID.simulate(q, event)
    elif ...:
```

(c) SystemVerilog
(FIFO impl & valid/ready omitted)

```
module IF(output inst);
  always @(posedge clk)
    inst <= mem[pc]; /*...*/
endmodule
```

```
module ID(
 input inst,
 output opcode, output data,
); /*...*/
endmodule
```

```
module top();
 logic inst_o;
 logic inst_i;
 IF fetcher(.inst(inst_o));
 StageFIFO if_id(
  .in(inst_o), .out(inst_i));
 ID decoder(
  .inst(inst_i),
  // Connect to EX stage
  .opcode(...), .data(...));
endmodule
```



Assassyn Design

Overview

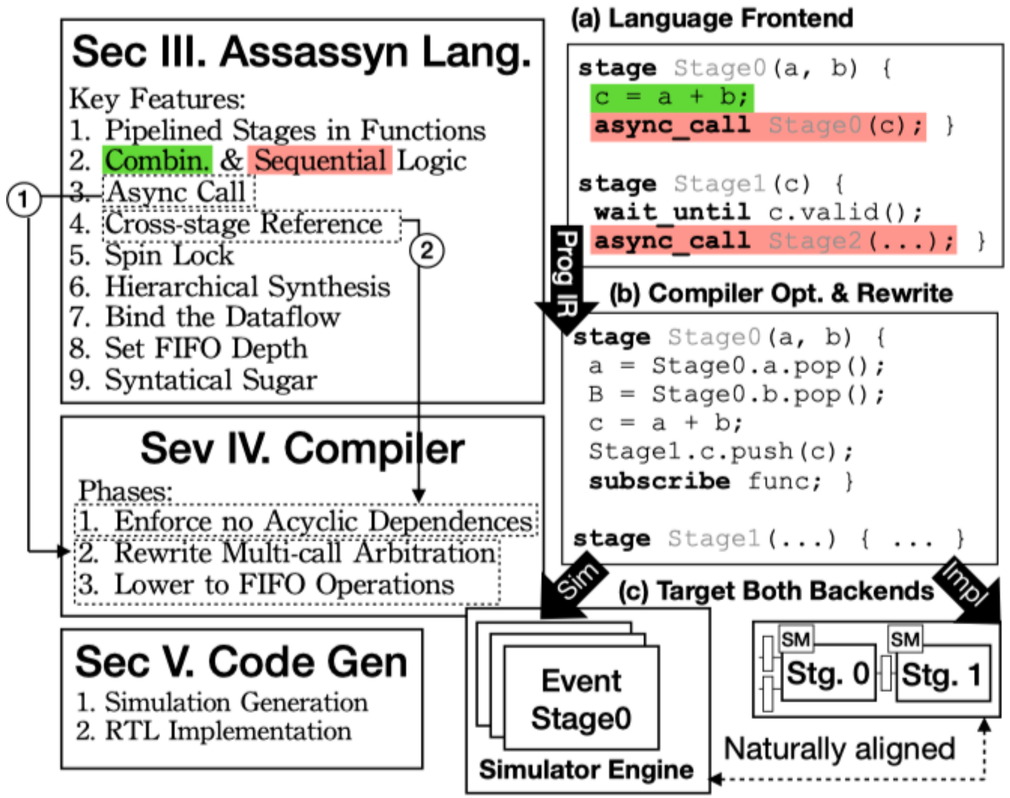
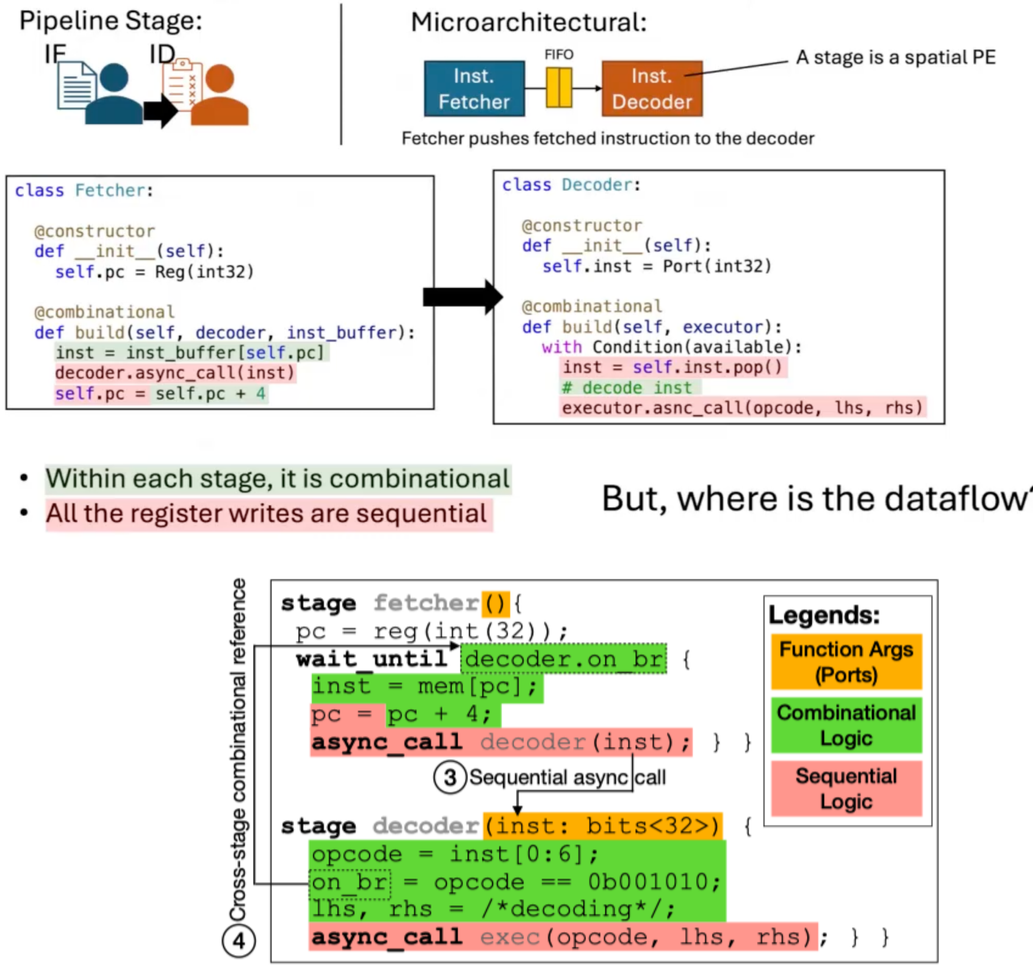


Figure 3: An overview to Assassyn.

Assassyn Design

A Boundary Combination and Sequential

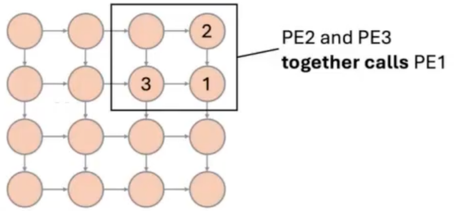
- How to constrcut pipeline?
  - Functions serve as the most fundamental building block of Assassyn to construct each stage of a pipelined architecture
- combinational: all arithmetic computations performed synchronously within this function are considered, which can be completed within one cycle
- sequential : any side-effect operations (such as register updates, and memory write)
- Invoke stages asynchronously



Assassyn Design

Bind the dataflow

- Function call is a single-chain dataflow, how to simulate the dataflow in hardware?
- Bind: introduce the concept of function bind
  - partial: It creates a new, simpler function by setting some of the original function's arguments in advance.
  - Call goo, only give it parameter b



- Software Programming
  - `foo(*args):args` are all in caller
- Hardware Design/Impl.
  - Data from A & B flow to C

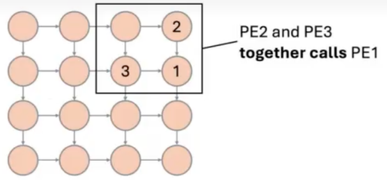
```
from functools import partial
def foo(a, b):
  return a + b
goo = partial(foo, 5)

# equivalent to foo(5, 10)
goo(10)
```

- Bind the dataflow
- There are too many “bind”s in Python, which one?

Assassyn Design

Bind the dataflow

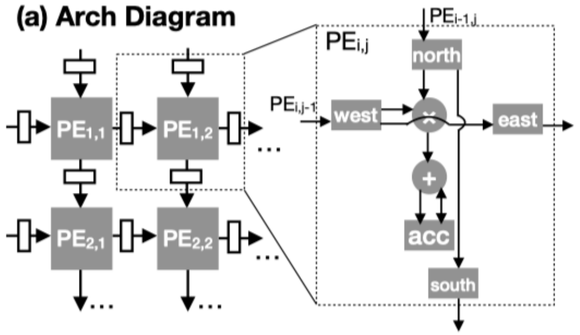
- PE1 needs to result of PE2 and PE3. PE2 and PE3 together calls PE1
- Module PE, each has the north and west input. When build the PE2, also generate the handle of PE1
- Dataflow
  - Async east(west=self.west). Pass self.west to the neiborgh PE.
  - Bind south(north=self.north). Bind the south neiborgh's north input with self.north
- In a word, use the function bind to construct multiple inputs



- Higher-order Function
  - A function that returns a function
  - Constructing pipeline stage is constructing a parameterized function

```
# module
class PE:
  def __init__(self):
    self.north = Port(8)
    self.west = Port(8)

  def build(south, east):
    async east(west=self.west)
    bind south(north=self.north)
```

```
# init pe[i][j] w/ PE()
# top function
for i in 1..5:
  for j in 1..5:
    pe[i][j].build(pe[i+1][j],
                   pe[i][j+1].bound)
```



(b) Higher-order Stage Function

Legend: Higher-order Arguments

```
1 [#static_timing]
2 stage PE
3   (east:PE,south:bind)
4   (west:int<32>,north:int<32>) {
5  acc = reg(int(32));
6  delta = west * north;
7  acc = acc + delta;
8  async_call east(west=west);
9  bound = bind south(north=north); }
```

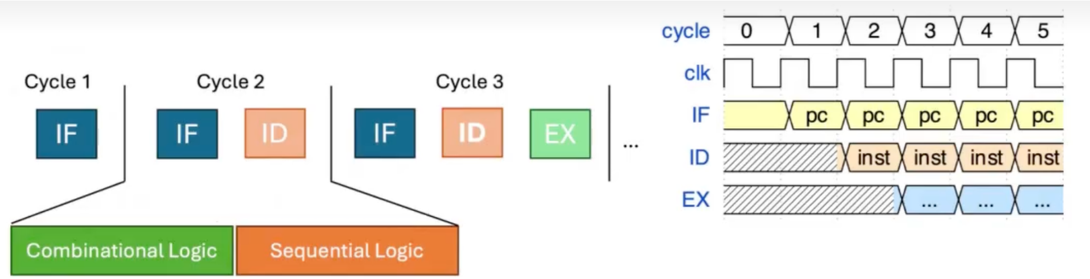
(c) Top Function Instantiation

```
1 # Assuming 1-based array, and we
2 # have sentinels for row5 & col5
3 sa = [PE() array of 5x5];
4 # Recursively connect each PE
5 for i in 1..4: for j in 1..4:
6  sa[i][j] = PE.build(sa[i+1][j],
7                      sa[i][j+1].bound)
8  sa[I][j].fifo_depth(west=1,north=1)
```

Assassyn Design

CodeGen, Simulator

- For every cycle, the simulation is divided into two phases:
  - stage execution: simulate the behaviors of each stage
  - register commitment: update the values of the registers.



- A Cycle is separated into 2 halves
  - 1st Half: Executing the Combinational Logic Functions
  - 2nd Half: Commit the side effect registers

Source: https://b23.tv/UpyKD92

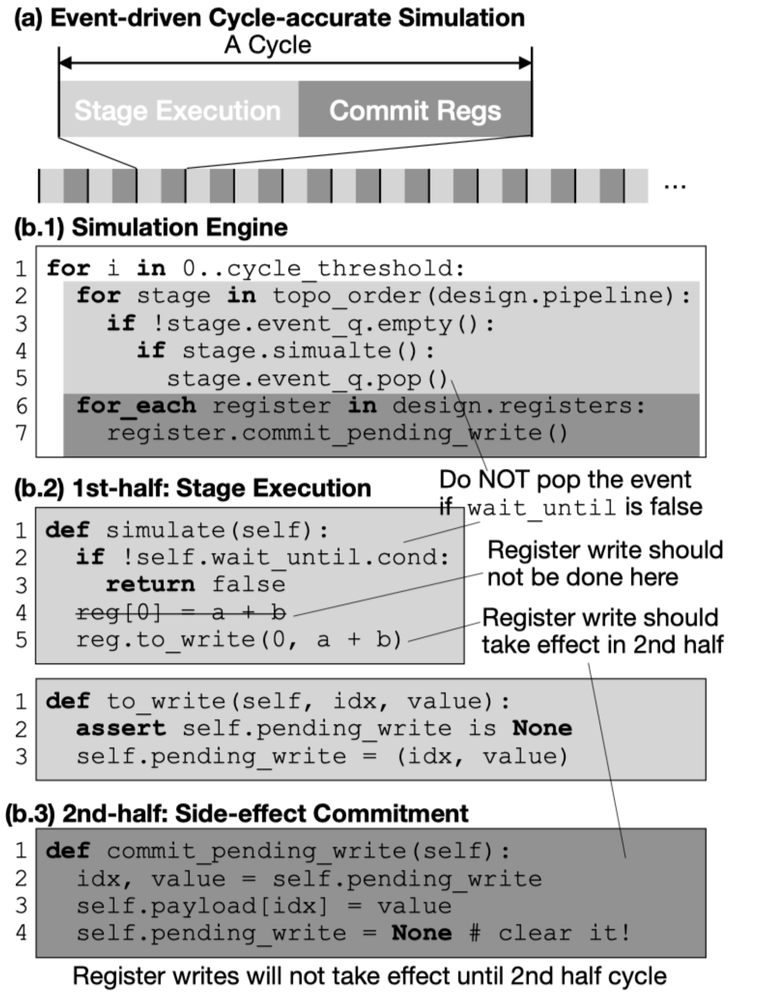
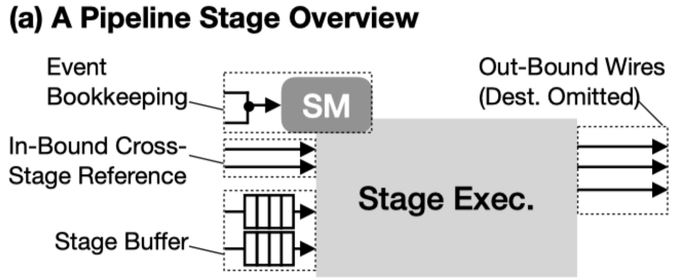


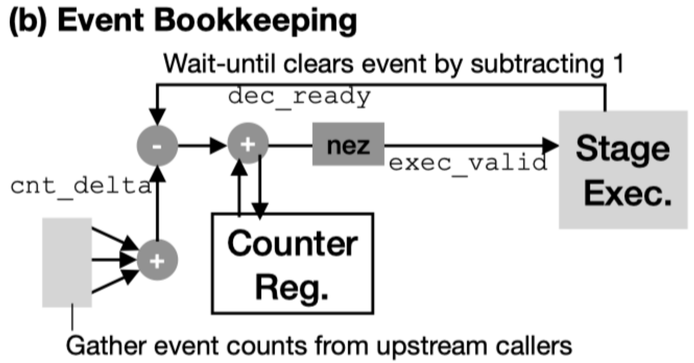
Figure 9: Assassyn-generated cycle-accurate simulator

Assassyn Design

CodeGen, RTL

- Each function is a pipeline stage
- Implicitly
  - All the staeg registers are FIFOs
  - A Counter SM is imposed to bookkeep executions
- Counter State Machine
  - Each function call increases
  - Each execution decreases
  - Counter != 0 enable stages execution



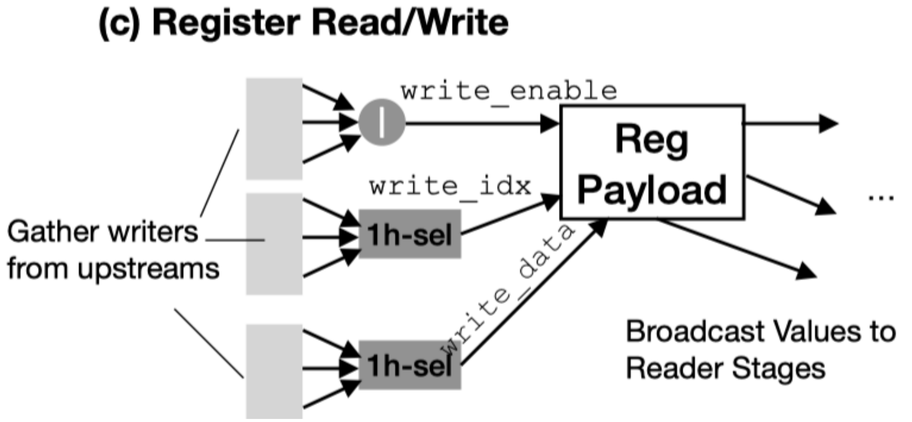


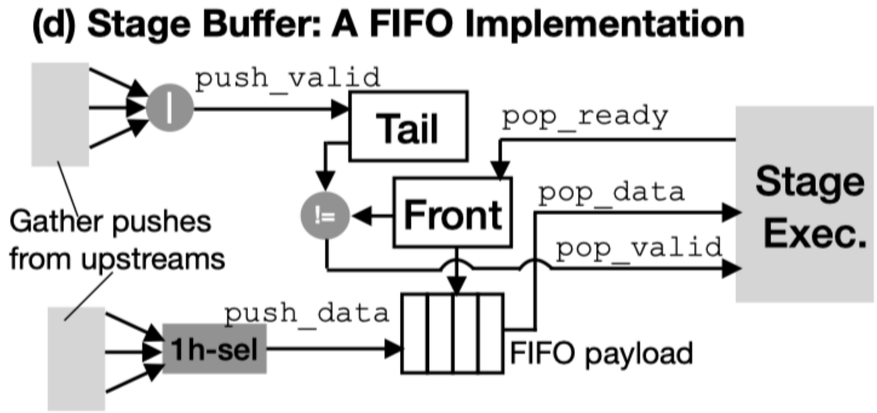
Source: https://b23.tv/UpyKD92

Assassyn Design

CodeGen, RTL

- Counter, Register and FIFO adopt a “gather and write paradigm
  - A key difference is that counter uses “+” to gather, while other 2 use “|”
- Registers are independent from each stage
- FIFO will fall back to registers when depth=1





Source: https://b23.tv/UpyKD92

Assassyn Evaluation

Evaluation, Lines of Code

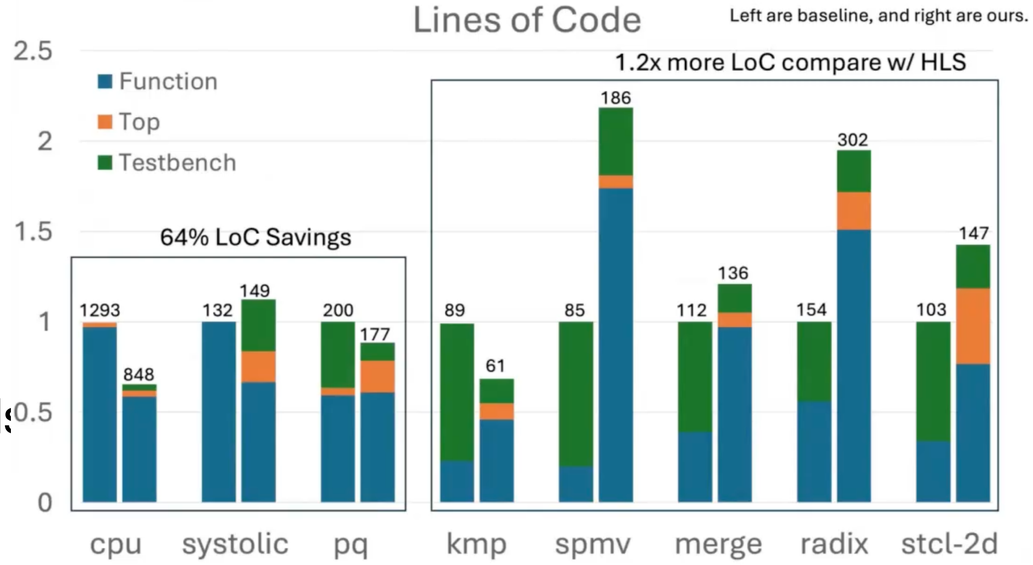
- Assassyn frontend, python; hardware intermediate representation is recorded in an abstract syntax tree (AST); Then this AST is fed to our backend (implemented in Rust) for compiler transformations and code generation
- Target design
  - Priority Q, CPU, Systolic array
  - Ellpack, stencil-2d,
- Lines of Code
  - RTL: Left, handcraft; right, Assassyn
  - HLS-generated Design: Assassyn-programmed workloads require 1.26× the LoC of the MachSuite C cod

Table 1: Manual Designs si

Target Design	Reference
Priority Q [12]	Manual Impl.
In-order CPU	Sodor [3]
Out-of-order CPU	
Systolic Array	Gemmini [28]

Table 2: HLS-generated Designs from MachSuite [58]

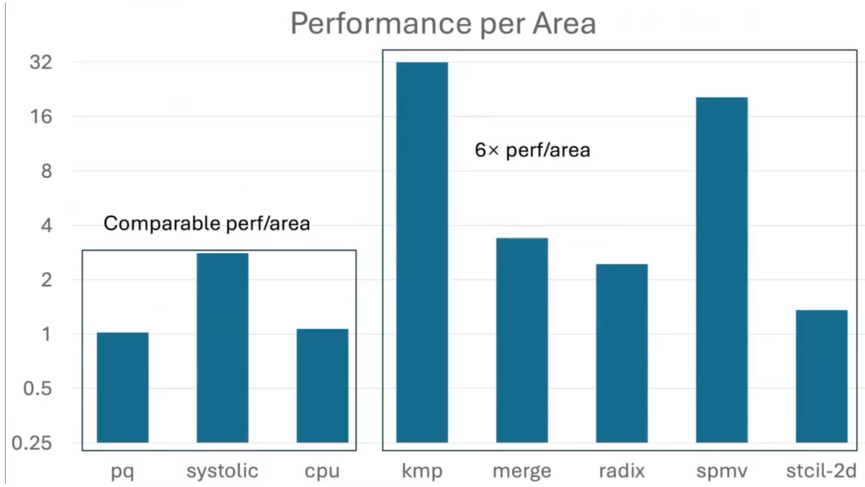
Application	Data Size
ellpack	n=494,m=10
stencil-2d	img=128^2,f=3^2
radix-sort	n=2048,m=16
kmp	n=32000,m=4
merge-sort	n=2048

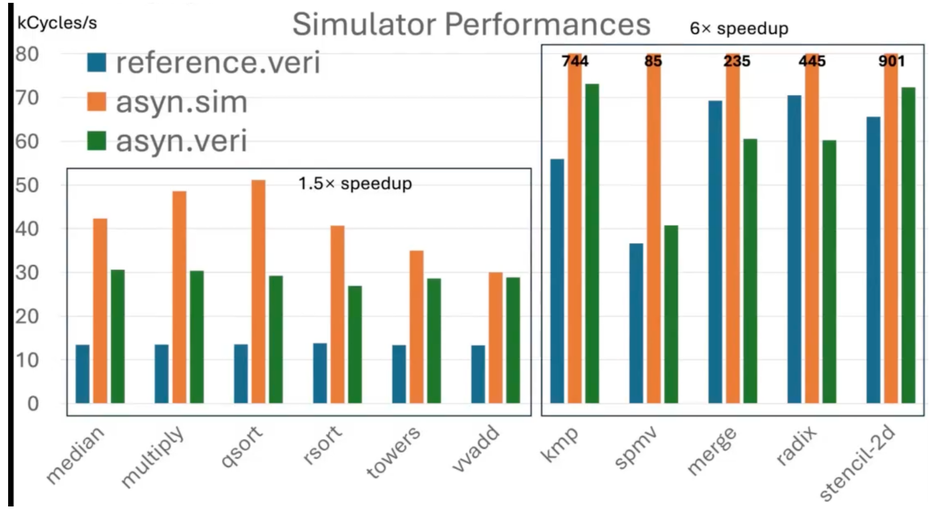


Assassyn Evaluation

Evaluation, Performance

- RTL, Performance per Area
  - Pq, cpu, 3-5%; The basedline Systolic array has extra units (activation), thus has larger area
- HLS, Performance per Area
  - 6x improvement compared to HLS
- Simulator, Speedup
  - Ref is instruction based granularity
  - Function based granularity, has less simulation time

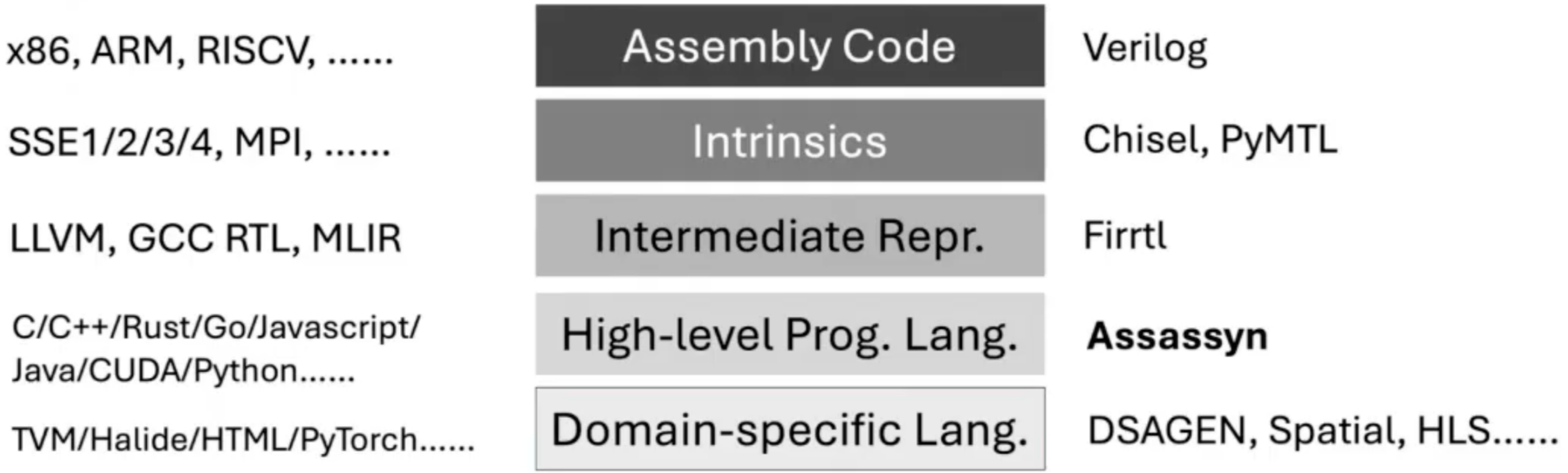




Assassyn Conclusion

A high-level Lang. for H/w Impl.

- Key Challenge: No high-level programming interfaces for hardware design and implementation



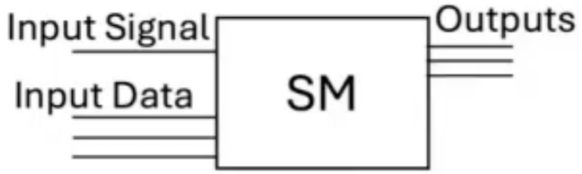
Source: https://b23.tv/UpyKD92

Assassyn Conclusion

Pull or Push?

- A key contradiction is the mind set between designing and implementing an architecture

A Verilog module **pulls** data in to advance its state machine when signal is enabled



IF **pushes** fetched instruction to ID

Conclusion: Trading-off Expressiveness

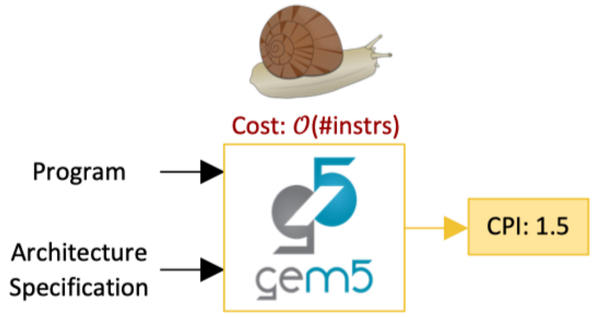
- Goto-statement is considered harmful
- For-loop & if-statement are enough for Turing completeness

Source: https://b23.tv/UpyKD92

Concorde, ISCA 2025

The Big Picture

Goal: Rather than simulate every μarchitecture detail, learn a *high-level* model that predicts performance, but runs *much faster*.



Five orders of magnitude faster

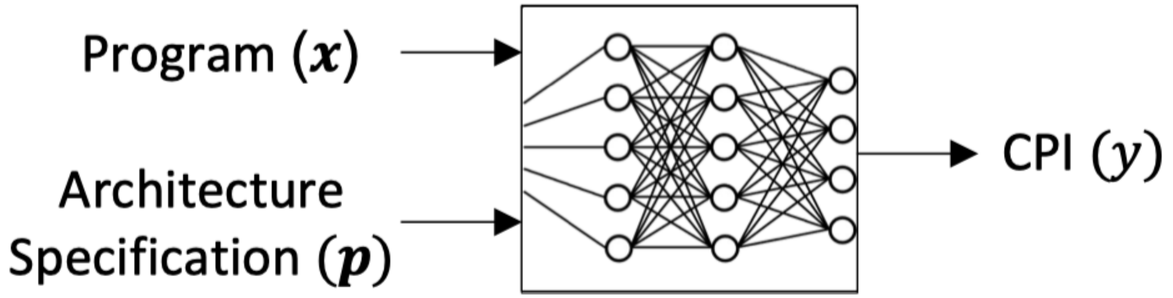
Only 2% CPI prediction error

3

Source: Concorde ISCA 2025 Slide

Concorde, ISCA 2025

Prior Work on Approximate Fast Simulation



Instruction-level

Cost: $\mathcal{O}$(#instrs)
Slow training
Slow inference

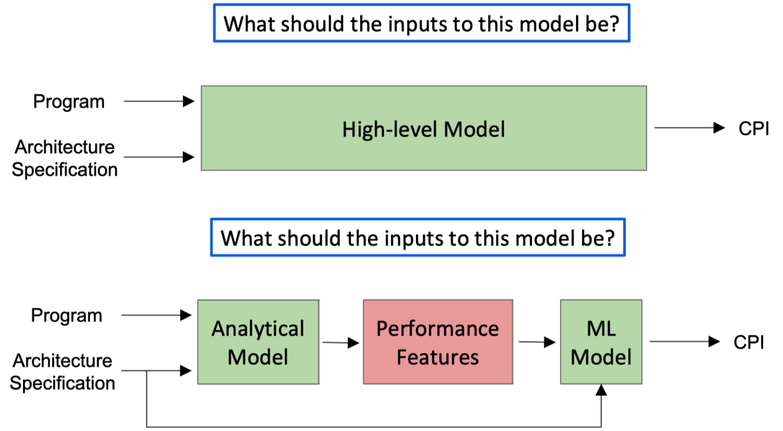
SimNet: Accurate and High-performance Architecture Simulation using Deep Learning, ACM SIGMETRICS/IFIP PERFORMANCE '22
TAO: Re-Thinking DL-based Microarchitecture Simulation, ACM SIGMETRICS/IFIP PERFORMANCE '24

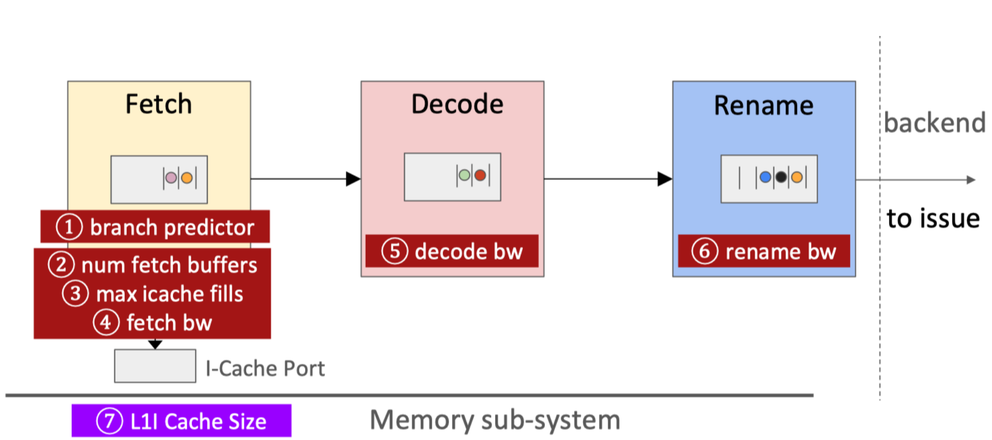
Source: Concorde ISCA 2025 Slide

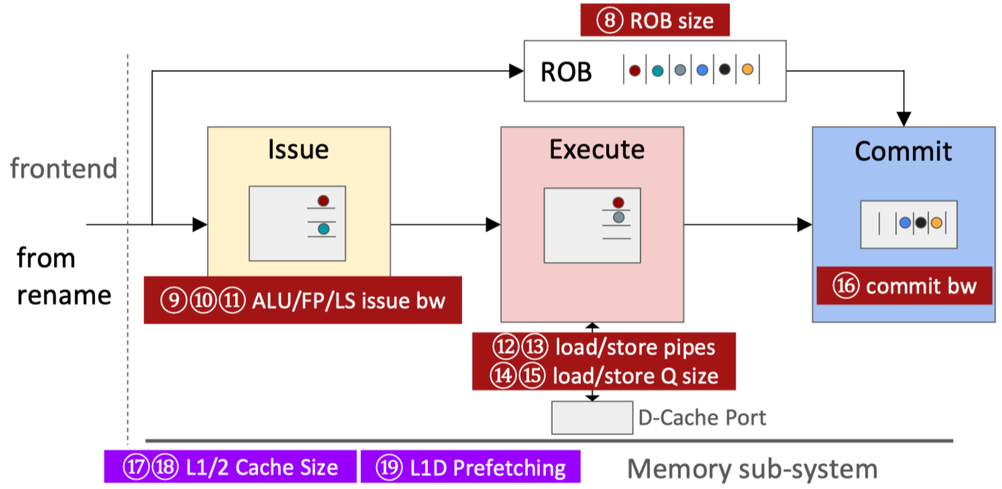
Concorde, Design

Overall Approach

- Analytical Model
  - Find the bottleneck of uarch component (ROB, decode bw) independently







Source: Concorde ISCA 2025 Slide

Concorde, Design

Key Idea: Per-Resource Throughput Analysis

- Set the different parameters of uarch component
- Example: set different ROB size, obtain the IPC

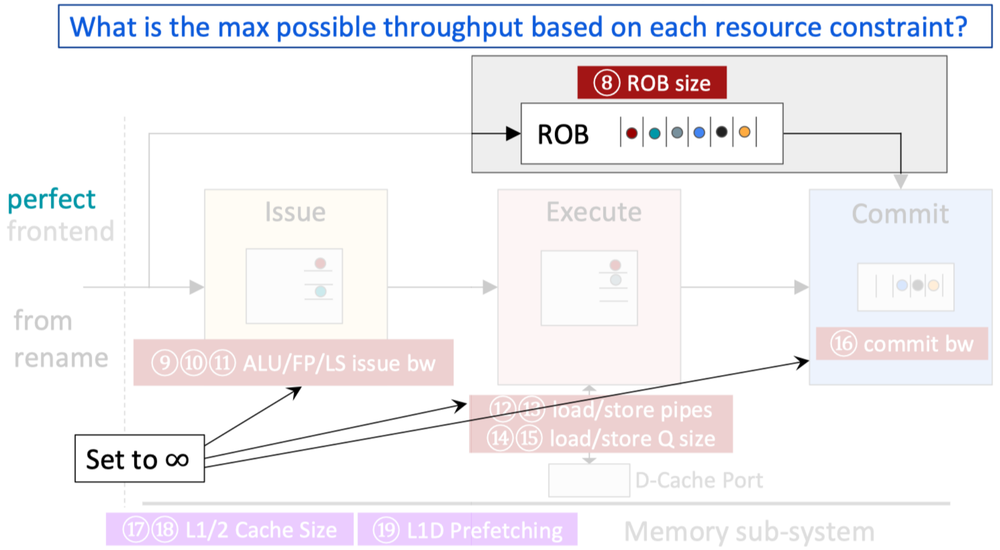


Table 1: Large space of design parameters

Parameter	Value Range	ARM N1 value
ROB size	1,2,3,...,1024	128
Commit width	1,2,3,...,12	8
Load queue size	1,2,3,...,256	12
Store queue size	1,2,3,...,256	18
ALU issue width	1,2,3,...,8	3
Floating-point issue width	1,2,3,...,8	2
Load-store issue width	1,2,3,...,8	2
Number of load-store pipes	1,2,3,...,8	2
Number of load pipes	0,1,2,...,8	0
Fetch width	1,2,3,...,12	4
Decode width	1,2,3,...,12	4
Rename width	1,2,3,...,12	4
Number of fetch buffers	1,2,3,...,8	1
Maximum I-cache fills	1,2,3,...,32	8
Branch predictor	Simple, TAGE	TAGE
Percent misprediction for *Simple* BP	0,1,2,...,100	–
L1d cache size (kB)	16,32,64,128,256	64
L1i cache size (kB)	16,32,64,128,256	64
L2 cache size (kB)	512,1024,2048,4096	1024
L1d stride prefetcher degree	0 (OFF),4 (ON)	0(OFF)

Source: Concorde ISCA 2025 Slide

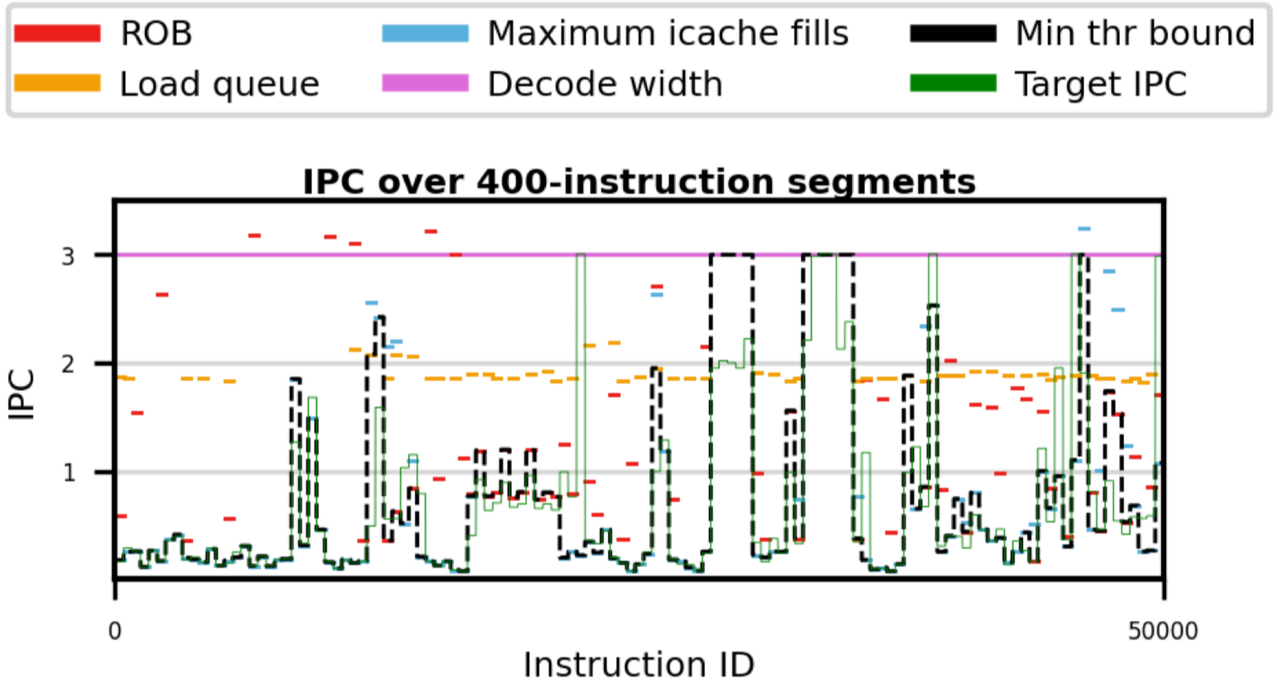
Concorde, Design

Key Idea: Per-Resource Throughput Analysis

- **Observation**: the min(throughput) of all compnents can reflect the tracy of IPC

- Instruction ID is a timeseries plots

Analytical model: In each window, $\mathrm{IPC} = \min_r(\mathrm{Throughput}_r)$

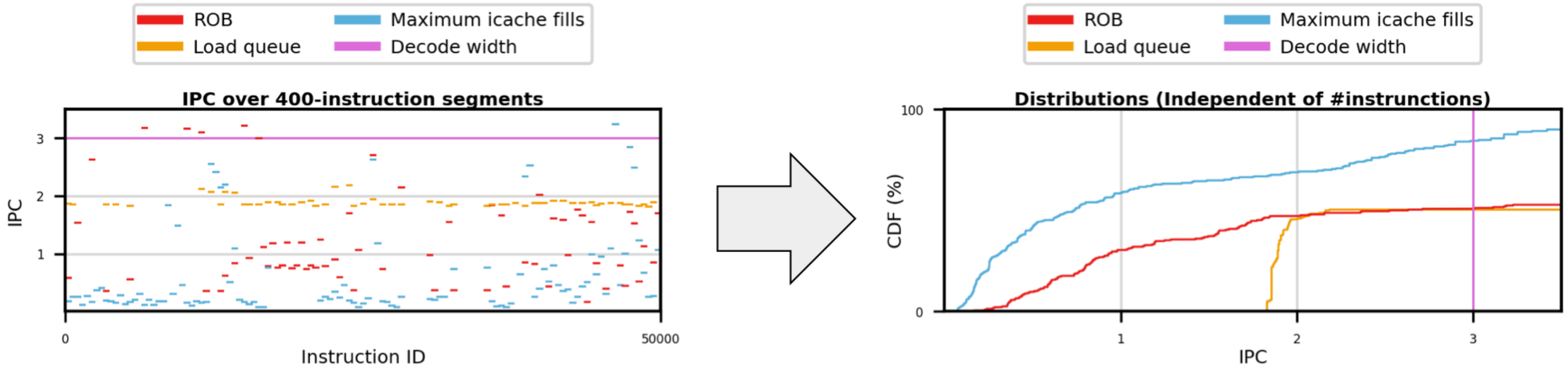


Source: Concorde ISCA 2025 Slide

Concorde, Design

Key Idea: Per-Resource Throughput Analysis

- **Feature Extraction:** CDF graph. Focus on IPC=1
  - Red line: ~30% instrutions, ROB makes the IPC <=1
  - Blue line: ~60% instrutions, icache makes the IPC <=1

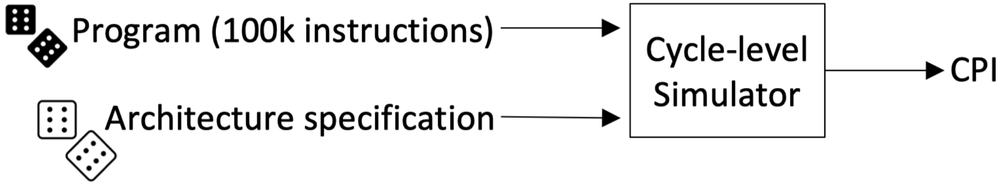


Source: Concorde ISCA 2025 Slide

Concorde, Design

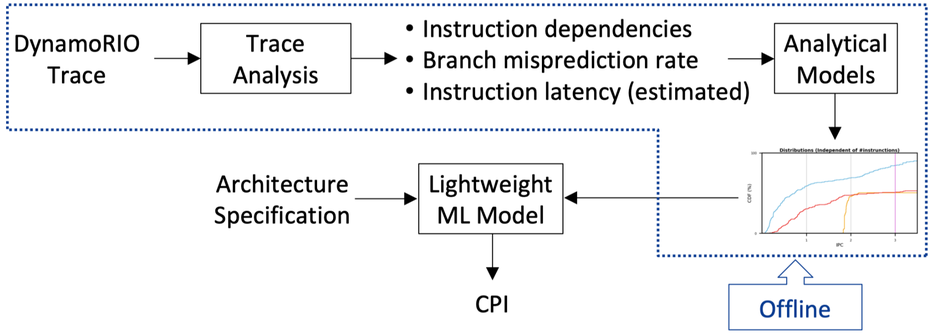
Putting it all together

- Generate the offline dataset, as the input of ML model



Parameter	Value Range
ROB size	1,2,3,...,1024
Commit width	1,2,3,...,12
Load queue size	1,2,3,...,256
Store queue size	1,2,3,...,256
ALU issue width	1,2,3,...,8
Floating-point issue width	1,2,3,...,8
Load-store issue width	1,2,3,...,8
Number of load-store pipes	1,2,3,...,8
Number of load pipes	0,1,2,...,8
Fetch width	1,2,3,...,12
Decode width	1,2,3,...,12
Rename width	1,2,3,...,12
Number of fetch buffers	1,2,3,...,8
Maximum I-cache fills	1,2,3,...,32
Branch predictor	Simple, TAGE
Percent misprediction for *Simple* BP	0,1,2,...,100
L1d cache size (kB)	16,32,64,128,256
L1i cache size (kB)	16,32,64,128,256
L2 cache size (kB)	512,1024,2048,4096
L1d stride prefetcher degree	0 (OFF),4 (ON)

Type	Name
Proprietary	Compression (P1) Search1 (P2) Search4 (P3) Disk (P4) Video (P5) NoSQL Database1 (P6) Search2 (P7) MapReduce1 (P8) Search3 (P9) Logs (P10) NoSQL Database2 (P11) MapReduce2 (P12) Query Engine&Database (P13)
Cloud Benchmark	Memcached (C1) MySQL (C2)
Open Benchmark	Dhrystone (O1) CoreMark (O2) MMU (O3) CPUtest (O4)
SPEC2017	505.mcf_r (S1) 520.omnetpp_r (S2) 523.xalancbmk_r (S3) 541.leela_r (S4) 548.exchange2_r (S5) 531.deepsjeng_r (S6) 557.xz_r (S7) 500.perlbench_r (S8) 525.x264_r (S9) 502.gcc_r (S10)

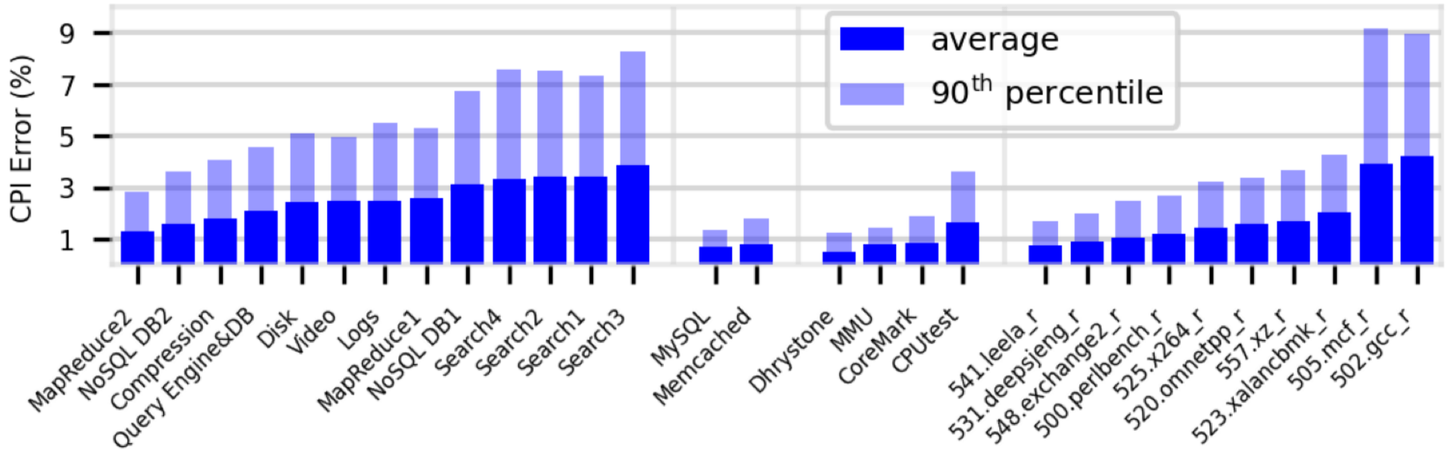


Source: Concorde ISCA 2025 Slide

Concorde, Design

Evaluation

- Benchmark: SPEC CPU 2017



Source: Concorde ISCA 2025 Slide

ISCA 2025

Others

- AMALI: An Analytical Model for Accurately Modeling LLM Inference on Modern GPUs

- Common Problem
  - Simulation-based Modeling: accurate, slow
  - Analytical Modeling: not accurate, specific
- What they cares? Accuracy, Simulation performance, Generality

Source: https://lights-sip-6j7.craft.me/IwKkKxKJdPCMBq
Source: https://notes.sjtu.edu.cn/BSVfVXK2SRighJD84Y93bw?view