Exploring Hardware/Software Co-Design Methodology

Billy Cai, Shruthi Ashwathnarayan, Farhan Shafiq,
Ahmed Eltantawy, Reza Azimi, Yaoqing Gao

Heterogeneous Compiler Lab,
Huawei Canada

Hardware-Software Co-Design: Pain Points

Time-to-market requirements :
Accelerated pace of hardware innovation requires a faster time-to-market for new chips

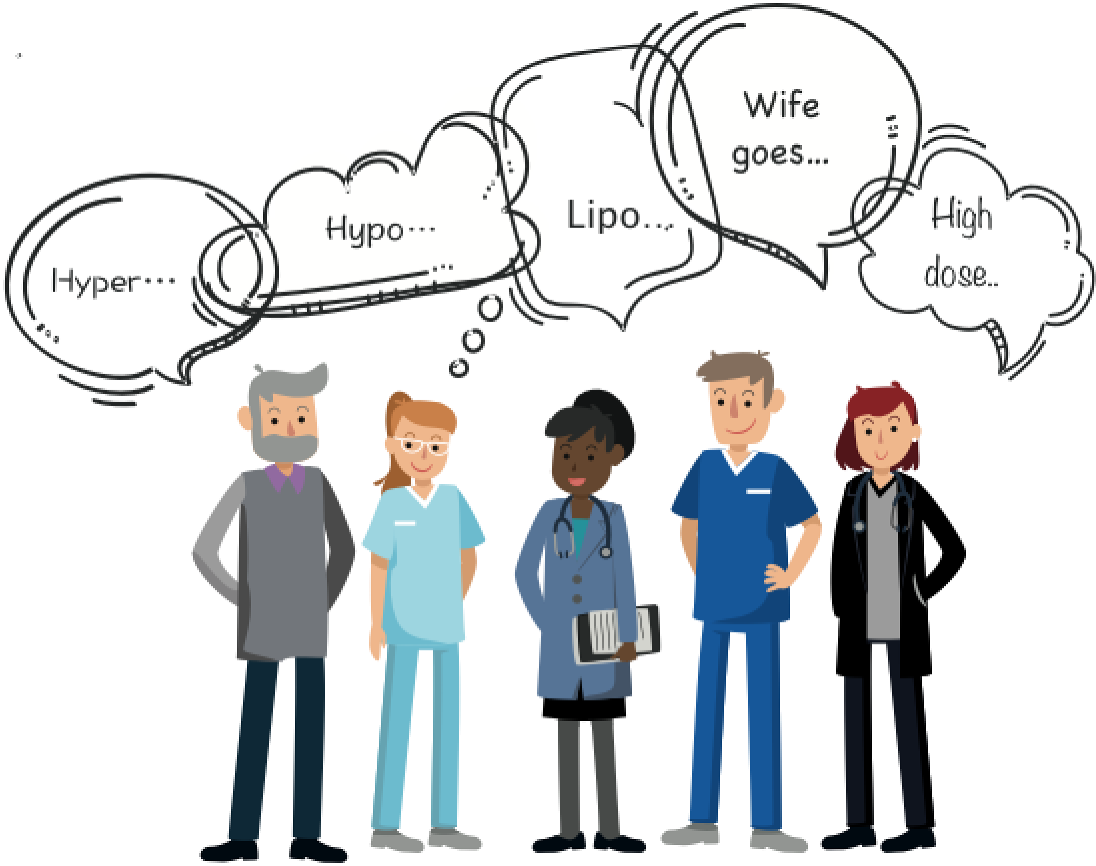
Complexity :
Size of complexity of hardware and software tools that need to be developed for each chip generation increases as design get more and more complex

Manual efforts :

- Lots of manual efforts, prone to delays and errors

Challenges of ISA Design

- Complexity of requirements
- Impacts on many interacting hardware and software components
  - Simulators
  - Compilers
  - Profilers
  - Synthesis Tools
  - Performance Libraries
- Multiple sources of “ground truth” – frequently at different versions and inconsistent
- Multiple interpretations due to the informal description (that could be possible inaccurate as it is not verifiable e.g., an instruction semantic).
- Lots of tedious manual work with every iteration of the ISA (and every new chip) to upgrade the tools to a new ISA version



www.chinadaily.com

Experience with Huawei’s AI Chip (Ascend)

- Multiple **generations** of the chip architecture have been designed
  - Design periods for different generations overlap in time
  - Major changes in the architecture and ISA between generations

- Multiple **variations** within the same generation
  - Targeting cloud, Mobile, IoT spaces
  - Different ISA subsets for each target
  - Hardware resources are also different (due to different performance requirements and different power and area budgets)

- Complex ISA
  - More than 200 instructions (scalar, vector, matrix, DMA)
  - Complex Instruction encoding semantics
  - Special instructions with complex semantics to accelerate very frequent NN patterns

- Huge Effort for Toolchain Development and Maintenance
  - Functional & Cycle Accurate Simulators
  - Assembler/Disassemblers
  - Optimizing Compiler Backends

Anecdotes from Chip X

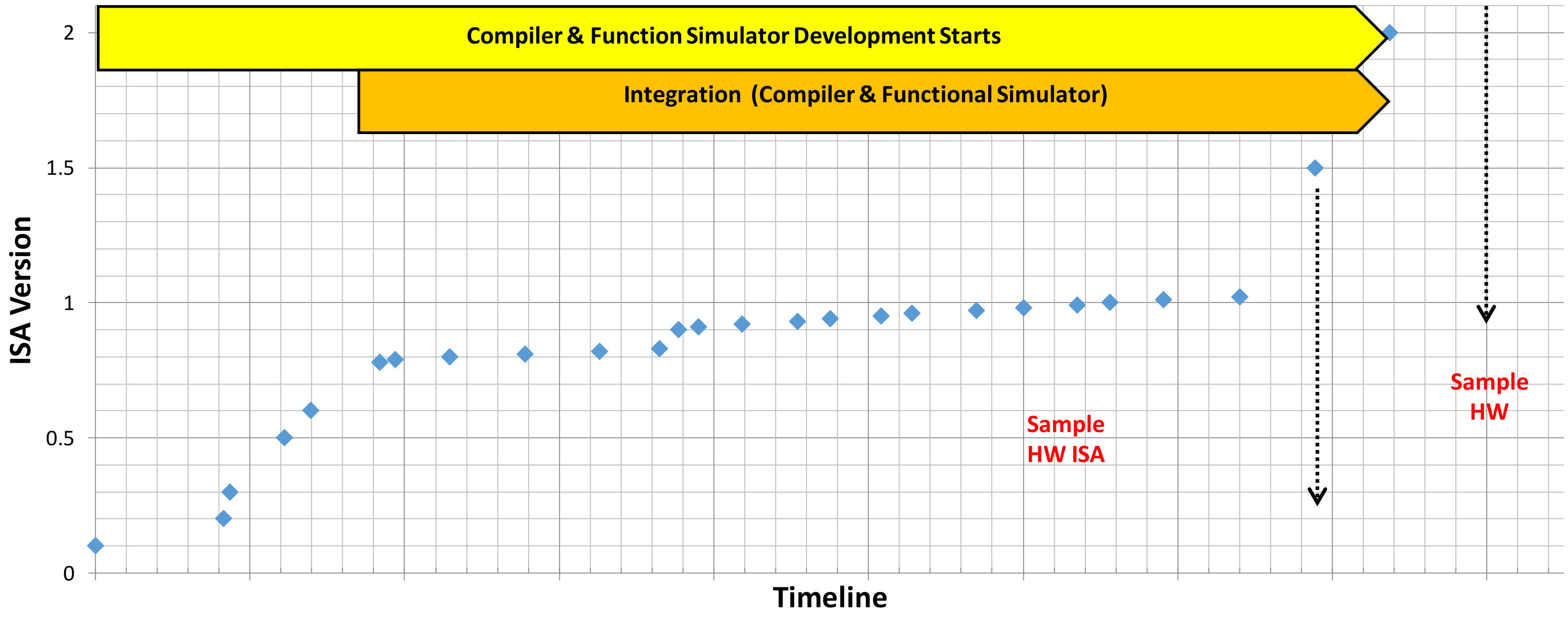
- SW (tool) development starts as early as ISA 0.1

- Integration tests are way before any major ISA release.

- 23 ISA iterations before major (relatively stable ISA release).

- Iterative development across compiler, simulators, RTL, testing continues as these ISA iterations comes up.

ISA updates in a course of a chip development



Retargetable HW/SW Co-design SDK

Goal:

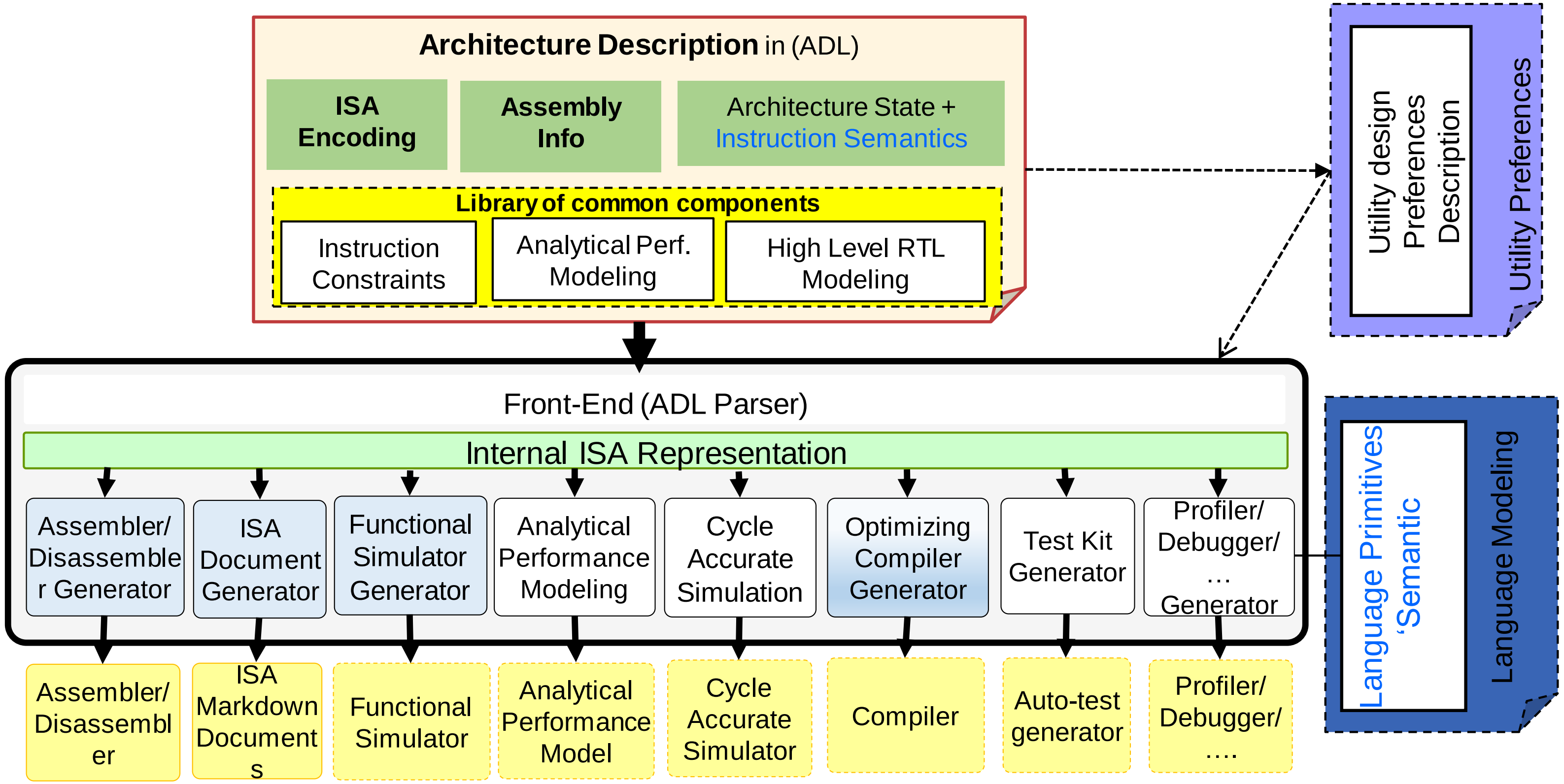
Develop a Huawei home-built framework for automatic generation of software development kit from a semi-formal single source description of Instruction Set Architecture.

ADL :

A single source description of ISA is maintained in an ADL (Architecture Description Language)

Output:

- Assembler / Disassembler
- ISA Markdown
- Functional simulator
- Compiler backend



Architecture Description in (ADL)

ISA Encoding

Assembly Info

Architecture State + Instruction Semantics

Library of common components

Instruction Constraints

Analytical Perf. Modeling

High Level RTL Modeling

Utility design Preferences Description

Utility Preferences

Front-End (ADL Parser)

Internal ISA Representation

Assembler/ Disassembler Generator

ISA Document Generator

Functional Simulator Generator

Analytical Performance Modeling

Cycle Accurate Simulation

Optimizing Compiler Generator

Test Kit Generator

Profiler/ Debugger/ … Generator

Language Primitives 'Semantic

Language Modeling

Assembler/ Disassembler

ISA Markdown Documents

Functional Simulator

Analytical Performance Model

Cycle Accurate Simulator

Compiler

Auto-test generator

Profiler/ Debugger/ ….

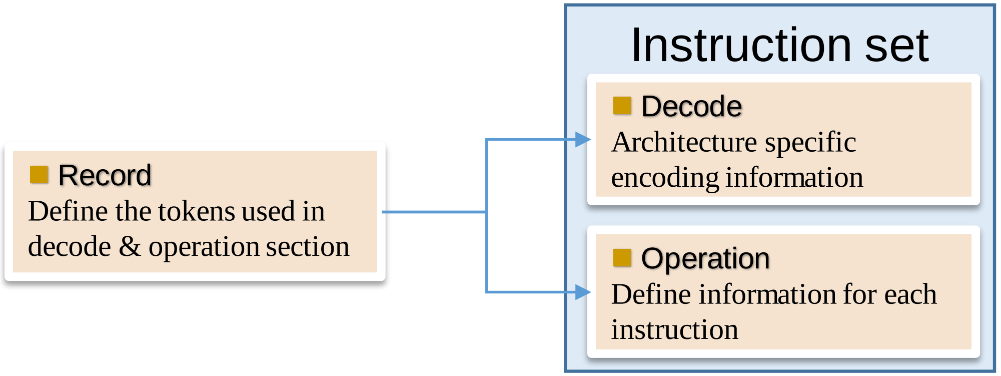
implemented

Future work

ADL Design Principles

- Retargetable ADL:
  - ADL is descriptive enough to represent various ISA types (e.g., variable and fixed instruction widths).
- Top-down Hierarchical representation of ISA
  - Concise representation
  - Easier to visualize, modify and verify
- Isolation of machine description from tools preferences
  - Enables HW designers to write the description
  - Unlike LLVM tablegen which mixes HW description with compiler design preferences (which can be written only by the compiler developers).
- Separation of back-end implementations:
  - Use an intermediate representation to represent ISA modeled by ADL.
  - Parse code once and reuse intermediate representation for different back-ends (compiler, simulator, markdiwn generation, etc).

ADL (further details, with RISC-V Examples)



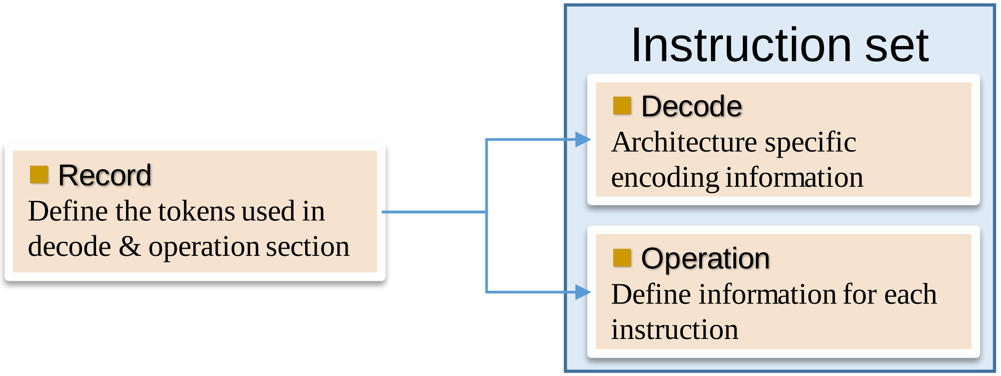
Instruction set

■ Decode
Architecture specific encoding information

■ Operation
Define information for each instruction

■ Record
Define the tokens used in decode & operation section

ADL (further details, with RISC-V Examples)



Instruction set

- Decode
  Architecture specific encoding information
- Operation
  Define information for each instruction

- Record
  Define the tokens used in decode & operation section

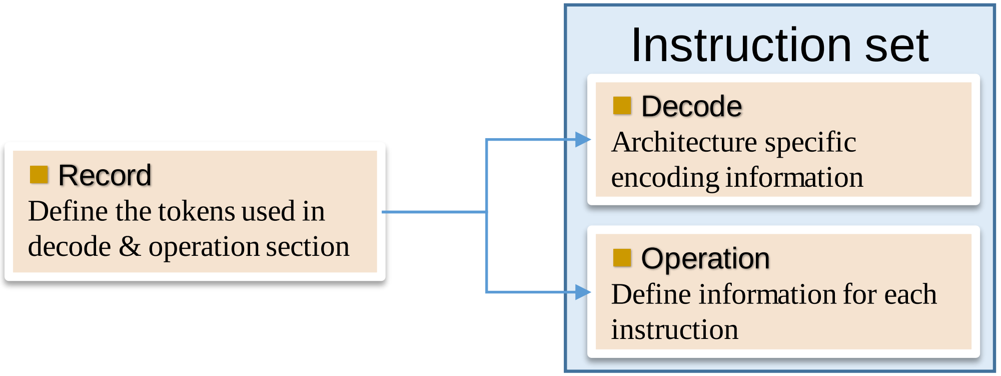
```
RECORD{
record GPR(
   "x"[0..31] = 5'd[0..31];
   "zero" = 5'd0;
   "ra" = 5'd1;
   "sp" = 5'd2;
     .
     .
     .
    )

GPR.set_alias(Xd,Xn,Xm,Xp);
}
```

Decode

```
RISCVInstr(
    decode : opcode? 7‘d3 { Base_I_type_load, opcode }
                                  7‘d35 { Base_S_type, opcode }
                                  7‘d19 { Base_I_type_imm, opcode }
)

RISCVInstr ::Base_I_type_load(
    decode : type? 3'd[0..5] { imm(12), Xm, type, Xd }
    encode(lw) : {type = 3'd2};
….
)

RISCVInstr ::Base_S_type(
    decode : type? 3'd[0..2] { imm[11:5], Xn, Xm, type, imm[4:0] }
    encode(sw) : {type = 3'd2 };
….
)

RISCVInstr ::Base_I_type_imm(
    decode : type? 3'd0 { imm(12), Xm, type, Xd }
    encode(addi) : {type = 3'd0 };
….
)
```

```
OPERATION{
//------BaseI_I_type_load--------
opn load(Xd, imm, Xm, type);
load.set_asm("lb %Xd,%imm(%Xm)“, type= 3’d0);
load.set_asm(“lh %Xd,%imm(%Xm)”, type= 3’d1);
load.set_asm(“lw %Xd,%imm(%Xm)”, type= 3’d2);
load.set_asm(“ld %Xd,%imm(%Xm)”, type= 3’d3);
…
//------BaseI_I_type_imm--------
opn addi(Xd, Xm, imm);
addi.set_asm("addi %Xd, %Xm, %imm");
addi.set_asm("mv %Xd, %Xm", imm = 12'd0);
…

//------BaseI_S_type--------------
opn sw(Xn, Xm, imm);
sw.set_asm("sw %Xn, %imm(%Xm)");
sw.set_description(“write value stored in Xn into memory”);
sw.set_behavior(“Mem[ val(Xm) + imm] = Xn”)
}
…
```

ADL (further details, with RISC-V Examples)

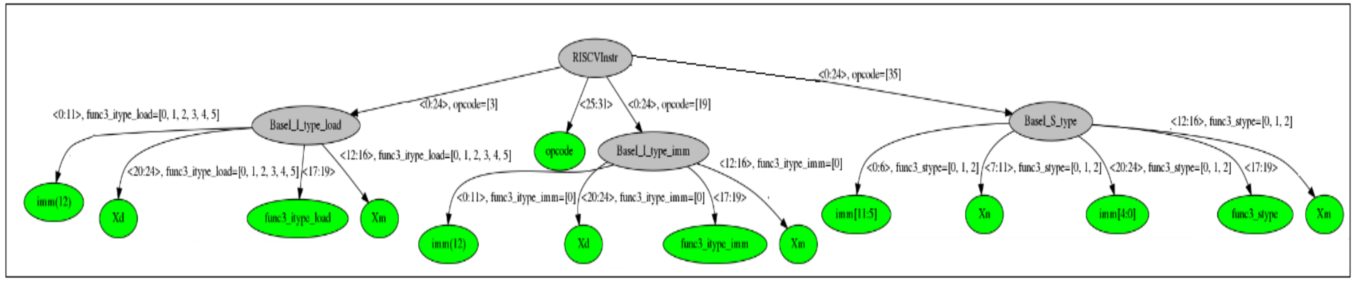


Instruction set

Decode
Architecture specific encoding information

Operation
Define information for each instruction

Record
Define the tokens used in decode & operation section



RISCVInstr
<0:24>, opcode=[3]
<25:31>
<0:24>, opcode=[19]
<0:24>, opcode=[35]
BaseI_I_type_load
opcode
BaseI_I_type_imm
BaseI_S_type
<0:11>, func3_itype_load=[0, 1, 2, 3, 4, 5]
<20:24>, func3_itype_load=[0, 1, 2, 3, 4, 5]
<17:19>
<12:16>, func3_itype_load=[0, 1, 2, 3, 4, 5]
imm(12)
Xd
func3_itype_load
Xm
<0:11>, func3_itype_imm=[0]
<20:24>, func3_itype_imm=[0]
<17:19>
<12:16>, func3_itype_imm=[0]
imm(12)
Xd
func3_itype_imm
Xm
<0:6>, func3_stype=[0, 1, 2]
<7:11>, func3_stype=[0, 1, 2]
<20:24>, func3_stype=[0, 1, 2]
<17:19>
<12:16>, func3_stype=[0, 1, 2]
imm[11:5]
Xn
imm[4:0]
func3_stype
Xm

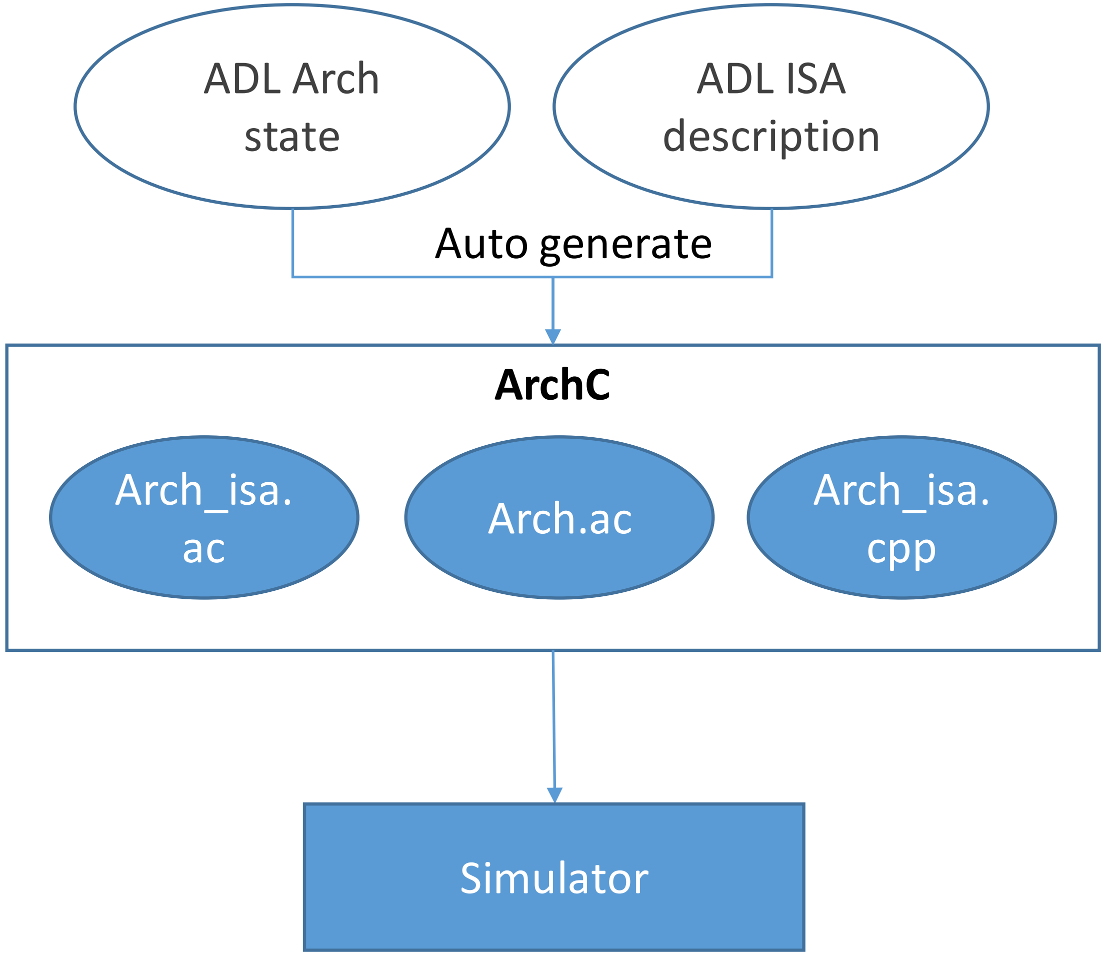
```
RECORD{
record GPR(
   "x"[0..31] = 5'd[0..31];
   "zero" = 5'd0;
   "ra" = 5'd1;
   "sp" = 5'd2;
    .
    .
    .
    )

GPR.set_alias(Xd,Xn,Xm,Xp);
}
```

```
RISCVInstr(
    decode : opcode? 7‘d3 { Base_I_type_load, opcode }
                              7‘d35 { Base_S_type, opcode }
                              7‘d19 { Base_I_type_imm, opcode }
)

RISCVInstr ::Base_I_type_load(
    decode : type? 3'd[0..5] { imm(12), Xm, type, Xd }
    encode(lw) : {type = 3'd2};
….
)

RISCVInstr ::Base_S_type(
    decode : type? 3'd[0..2] { imm[11:5], Xn, Xm, type, imm[4:0] }
    encode(sw) : {type = 3'd2 };
….
)

RISCVInstr ::Base_I_type_imm(
    decode : type? 3'd0 { imm(12), Xm, type, Xd }
    encode(addi) : {type = 3'd0 };
….
)
```

```
OPERATION{
//------BaseI_I_type_load--------
opn load(Xd, imm, Xm, type);
load.set_asm("lb %Xd,%imm(%Xm)“, type= 3’d0);
load.set_asm(“lh %Xd,%imm(%Xm)”, type= 3’d1);
load.set_asm(“lw %Xd,%imm(%Xm)”, type= 3’d2);
load.set_asm(“ld %Xd,%imm(%Xm)”, type= 3’d3);
…
//------BaseI_I_type_imm--------
opn addi(Xd, Xm, imm);
addi.set_asm("addi %Xd, %Xm, %imm");
addi.set_asm("mv %Xd, %Xm", imm = 12'd0);
…

//------BaseI_S_type---------------
opn sw(Xn, Xm, imm);
sw.set_asm("sw %Xn, %imm(%Xm)");
sw.set_description(“write value stored in Xn into memory”);
sw.set_behavior(“Mem[ val(Xm) + imm] = Xn”)
}
…
```

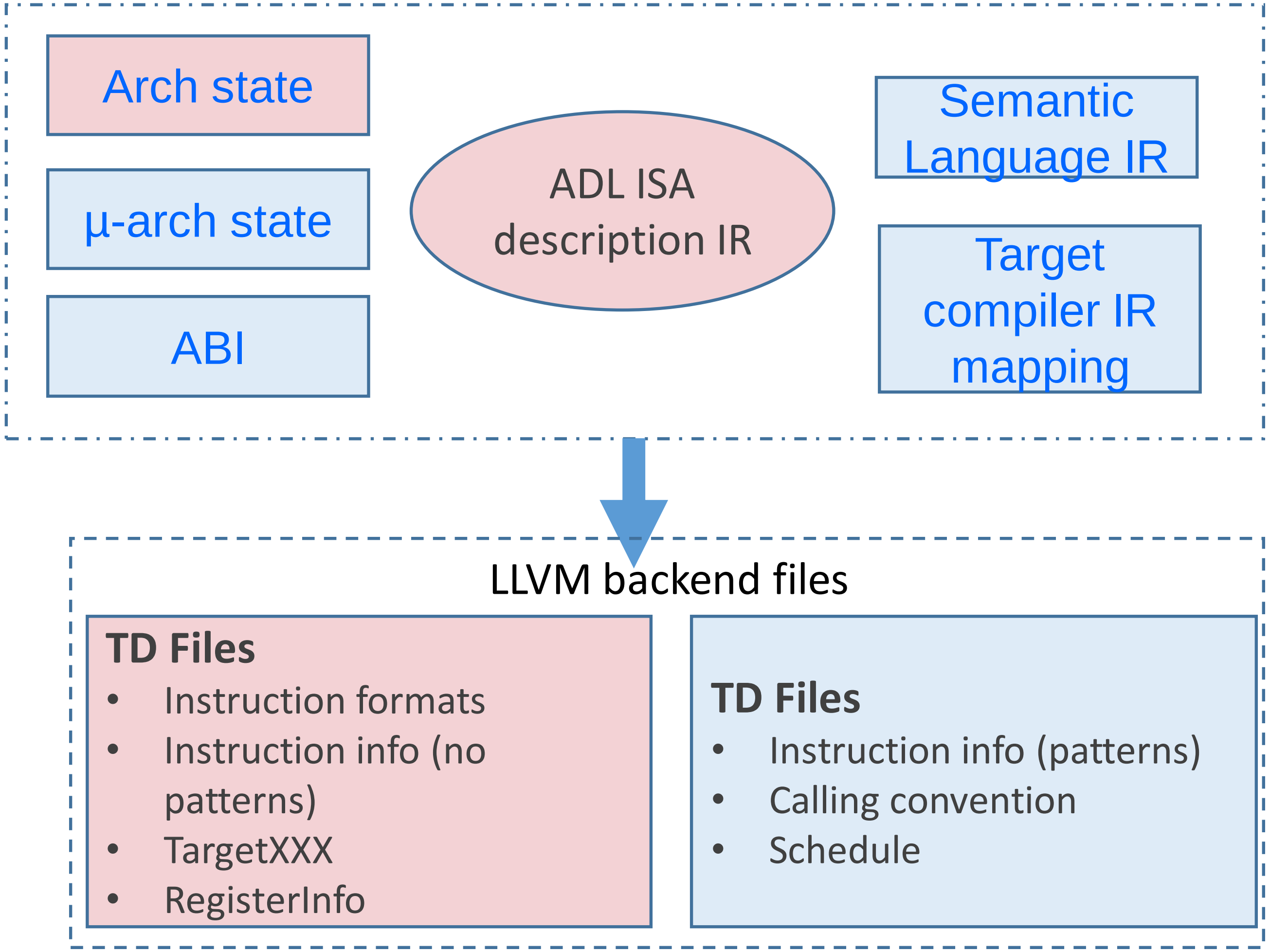
Auto-Generation of Functional Simulation

- ADL allows for Instruction behavior to be directly written in C code.
- ADL semantic description language is coupled with a behavior expressed in C.
- This can be used to have an auto generated simulator directly from the ADL
- Alternatively an existing simulator generator can be used by auto-generating required files.
- We generated a simulator using the ArchC framework



Auto-Generation of Compiler Backend

- ADL parser converts ADL description into an IR.
- RSDK compiler back-end generation tool uses IR and other architecture and compiler information to generate target backend files
- ADL IR and Arch state are the primary input files. Backend files can be generated independent of other input files, with hints on missing pieces.
- LLVM backend file generation tool was developed.
- RISCV architecture was used to demonstrate the tool



TD Files
- Instruction formats
- Instruction info (no patterns)
- TargetXXX
- RegisterInfo

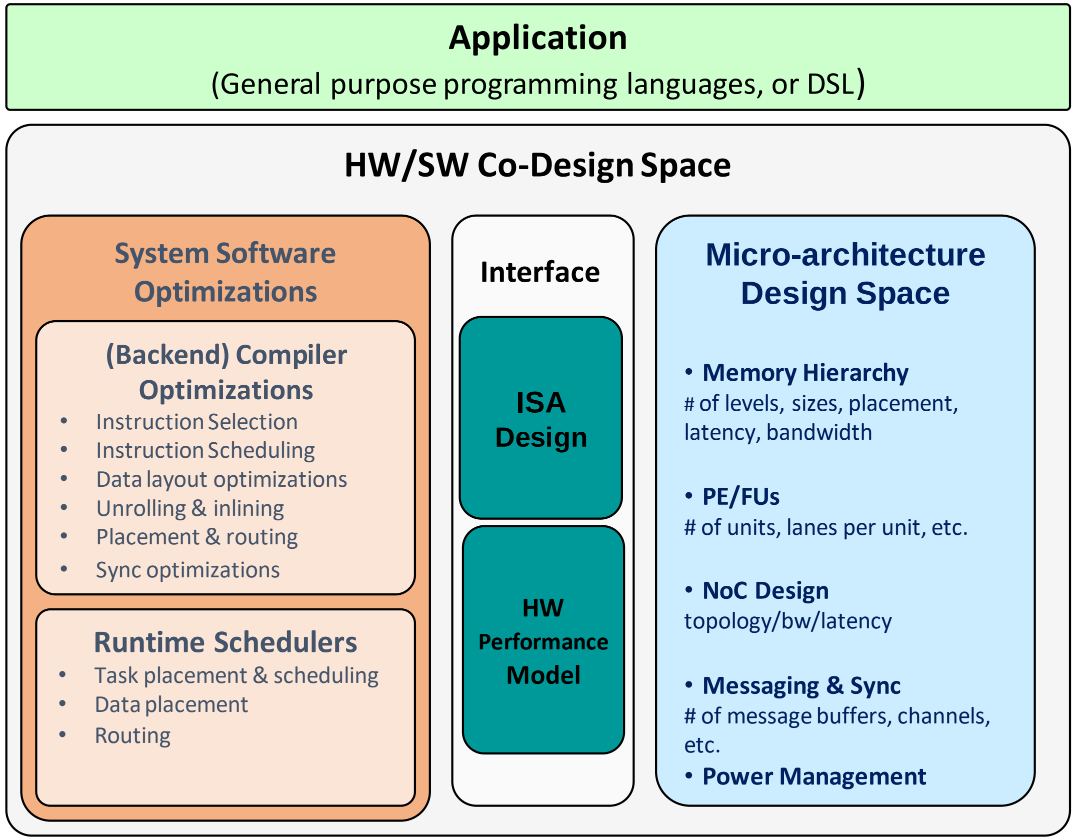
TD Files
- Instruction info (patterns)
- Calling convention
- Schedule

Semantic description Challenges

- A formal description of ISA semantics is needed for auto-generation of instruction selection.
- Semantic description needs to be simple and intuitive enough for ISA architects but also lend itself to compiler generation. Available formal description languages / tools are either too complicated from architect's perspective or not suitable for compiler generation.
- A generic IR Library was defined which is generic enough to represent various machine operations, while being very close to **LLVM's Generic Machine IR (GMIR)** so it can be correlated with compiled code's IR.
- RISC-V base ISA was described using the Library and selection patterns were generated.
- Faced few challenges including compiler pseudo instructions (e.g. `return`), target specific multi-class patterns (`load/store`), complex instruction behavior (`bit manipulation`) etc.

Challenge: Abstract Performance Modeling

- System performance is a function of both hardware and compiler
  - Hardware design space exploration is often conducted in isolation based on theoretical performance.
  - Performance of real applications is often different from HW theoretical peaks.
- Cycle-accurate simulation for thorough evaluation of different designs is impractical.
  - There are a huge number of designs in HW or SW design space.



Application

(General purpose programming languages, or DSL)

HW/SW Co-Design Space

System Software Optimizations

(Backend) Compiler Optimizations

- Instruction Selection
- Instruction Scheduling
- Data layout optimizations
- Unrolling & inlining
- Placement & routing
- Sync optimizations

Runtime Schedulers

- Task placement & scheduling
- Data placement
- Routing

Interface

ISA Design

HW Performance Model

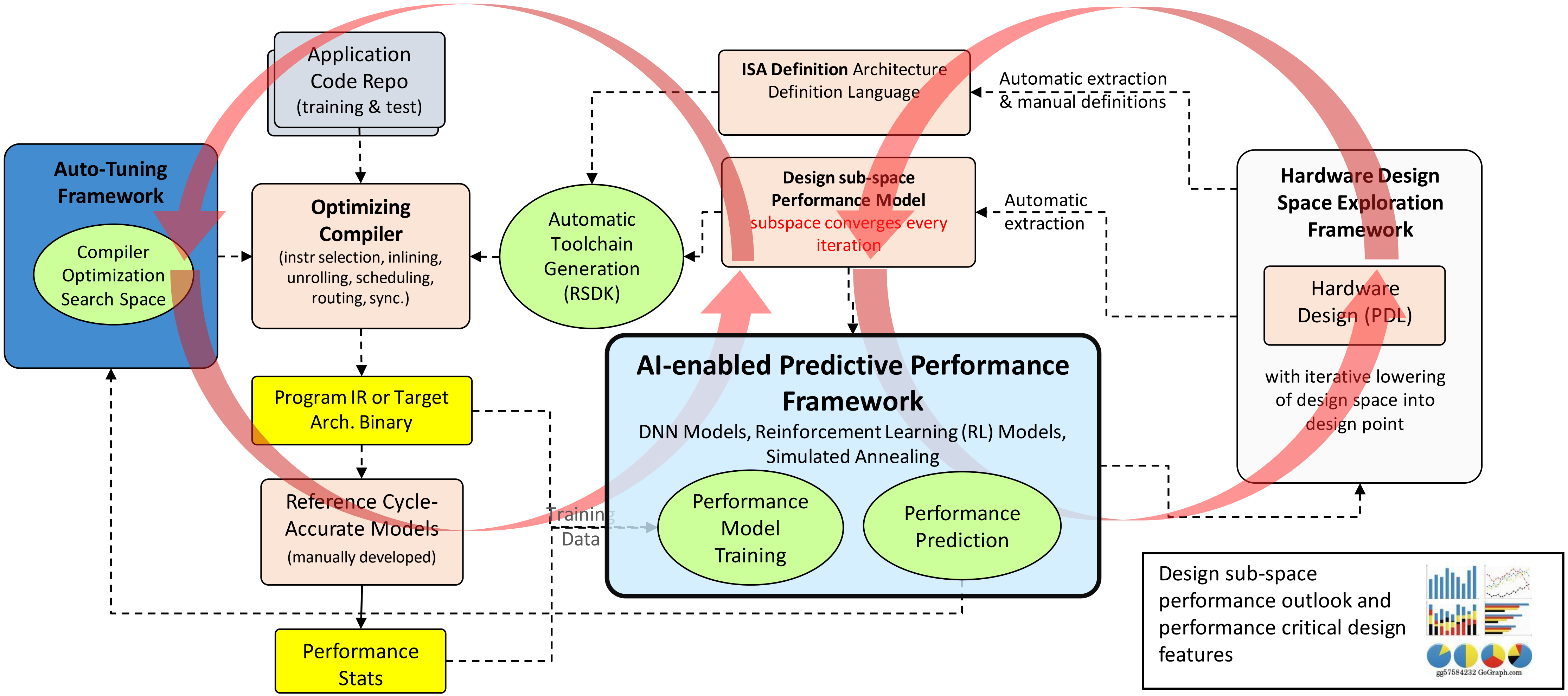
Micro-architecture Design Space

- **Memory Hierarchy**
  # of levels, sizes, placement, latency, bandwidth
- **PE/FUs**
  # of units, lanes per unit, etc.
- **NoC Design**
  topology/bw/latency
- **Messaging & Sync**
  # of message buffers, channels, etc.
- **Power Management**

Research on AI-based Approach to Performance Modeling

- **Predictive HW Design Exploration Tools**
  - Using machine-learning to efficiently explore the architecture/compiler co-design space
  - Efficiently Exploring Architectural Design Spaces via Predictive Modeling
  - Practical Design Space Exploration
  - Time loop/Accelergy: Tools for Evaluating Deep Neural Network Accelerator Designs [MIT/Stanford/NVIDIA]
- **ML-based (Cross-architecture) Performance Prediction**
  - Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks
  - Cross-architecture performance prediction
- **ML-based NoC Performance Prediction**
  - Machine Learning Based Framework to Predict Performance Evaluation of On-Chip Networks
  - UPM-NoC: Learning Based Framework to Predict Performance Parameters of Mesh Architecture in On-Chip Networks
- **AI for Guiding Compiler Optimizations**
  - Machine Learning in Compiler Optimizations
  - Graph-Based Deep Learning for Program Optimization & Analysis
  - End-to-end Deep Learning of Optimization Heuristics
- **Predictive AI/ML-based Autotuning**
  - A Survey on Compiler Autotuning using Machine Learning

Vision: AI-enabled Predictive Performance Modeling for Compiler-in-the-Loop Co-Design



Application Code Repo (training & test)

Auto-Tuning Framework

Compiler Optimization Search Space

Optimizing Compiler (instr selection, inlining, unrolling, scheduling, routing, sync.)

Automatic Toolchain Generation (RSDK)

ISA Definition Architecture Definition Language

Automatic extraction & manual definitions

Design sub-space Performance Model subspace converges every iteration

Automatic extraction

Hardware Design Space Exploration Framework

Hardware Design (PDL)

with iterative lowering of design space into design point

Program IR or Target Arch. Binary

Reference Cycle-Accurate Models (manually developed)

Training Data

Performance Stats

AI-enabled Predictive Performance Framework

DNN Models, Reinforcement Learning (RL) Models, Simulated Annealing

Performance Model Training

Performance Prediction

Design sub-space performance outlook and performance critical design features

Thank you