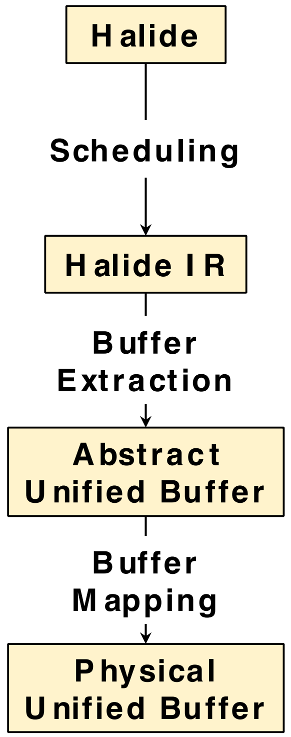
Compiling Halide Programs to our CGRA

Jeff Setter

Halide application → CGRA hardware

```
// Algorithm
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
// Schedule
blur.in().tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur.in(), xo)
        .compute_at(blur.in(), xo);
input.stream_to_accelerator();
```



Unified Buffer: Compiling Halide Programs to Push-Memory Accelerators

Qiaoyi (Joey) Liu, Dillon Huff, Jeff Setter, Maxwell Strange,

Mark Horowitz, Priyanka Raina, Fredrik Kjolstad

Introduction

Motivation

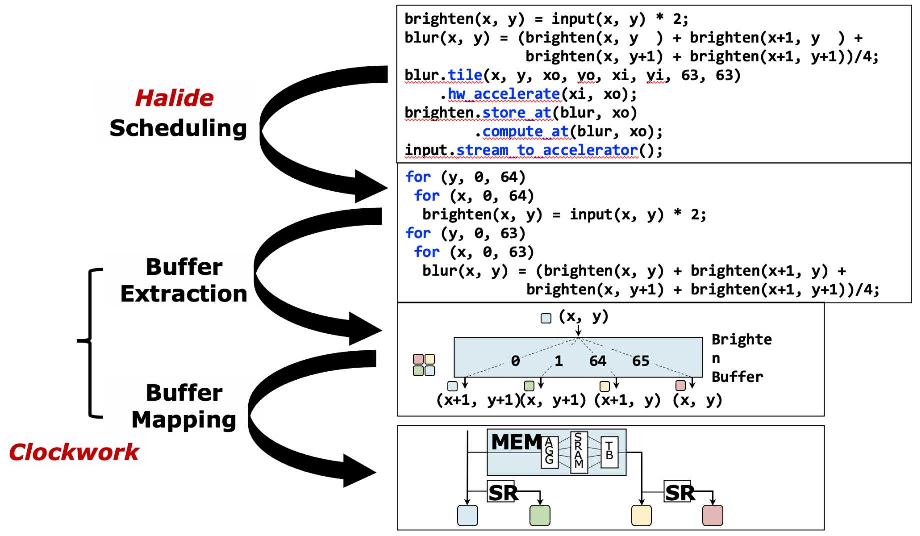
- Accelerators use highly optimized push memories
- Many efforts describe **memory designs**, but not how to **compile to memory**

Challenges

- Difficult to **generalize** the mapping process
    - Frontend algorithm: Stencil + DNNs
    - Backend architecture: HLS, library-based method
- Hard to **optimize** the compile result:
    - Large design space
    - Manual effort for a specific application

Solution: the Unified Buffer

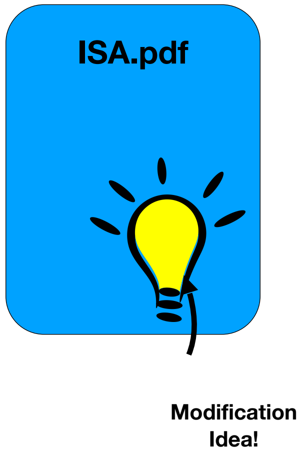
- Create an abstraction
  - Describe the information ***when*** and ***where*** data flows
  - Bundled with memory ***port***
  - In terms of ***operations***
- Leverage compiler Optimization
  - Polyhedral Analysis
  - Vectorization

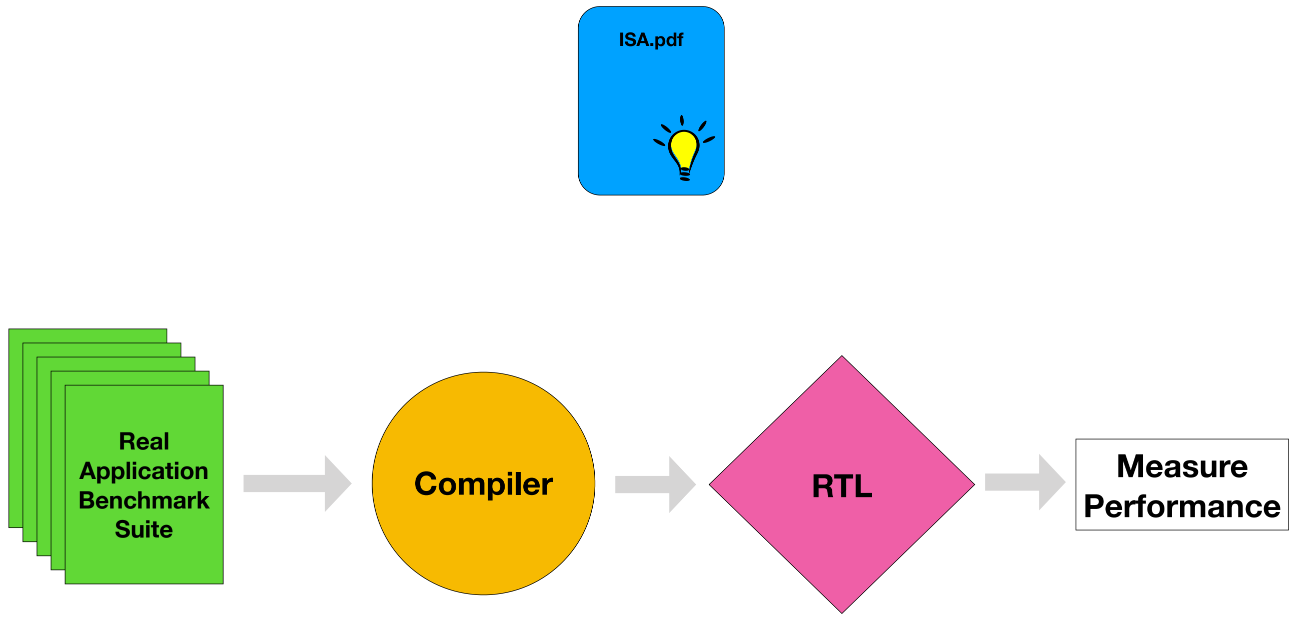


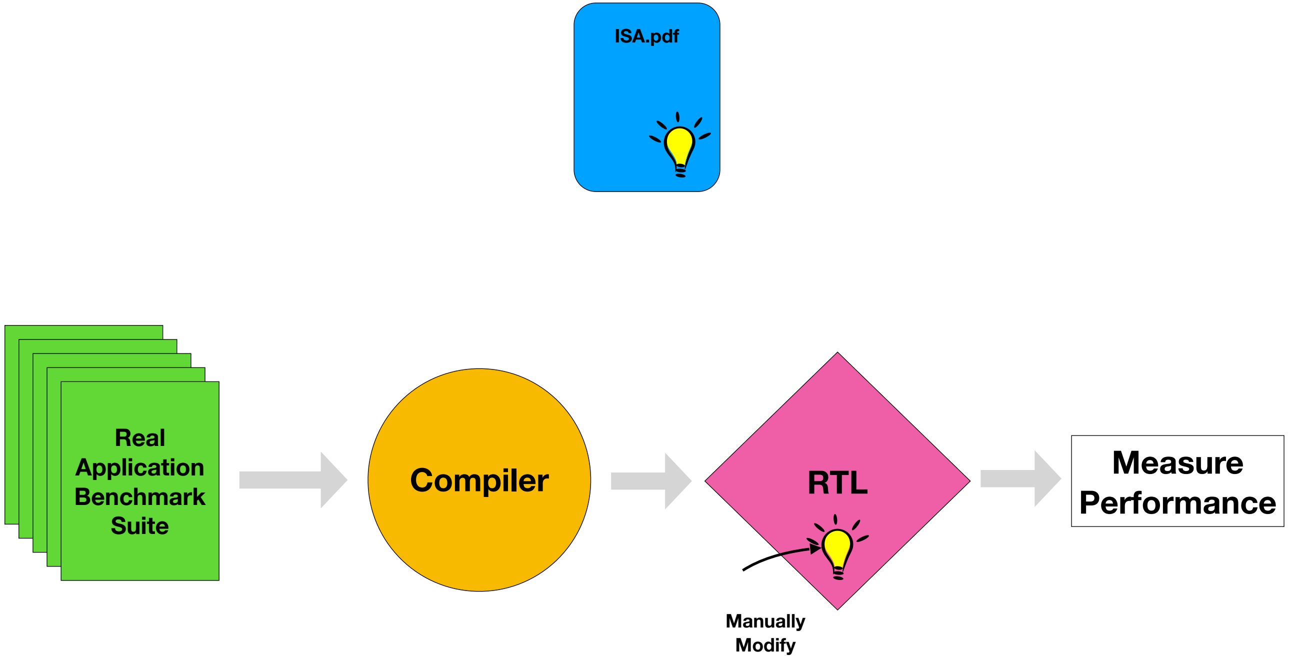
```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

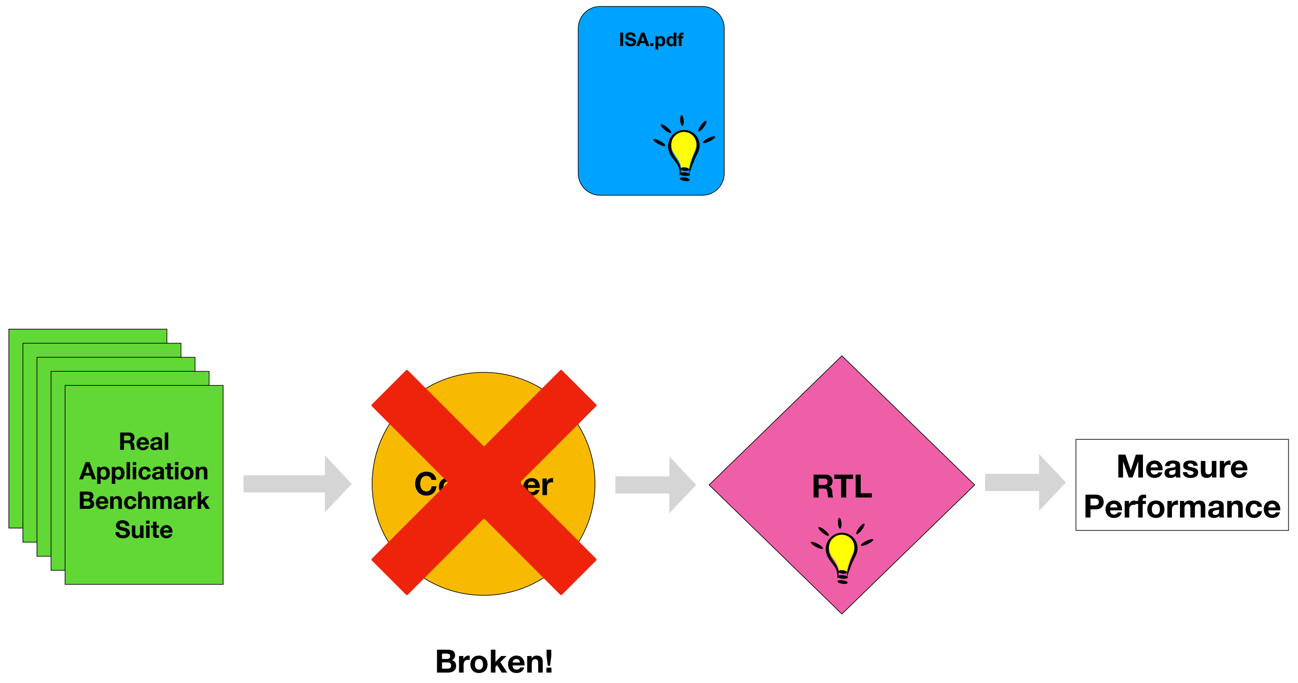
```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;
for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
```

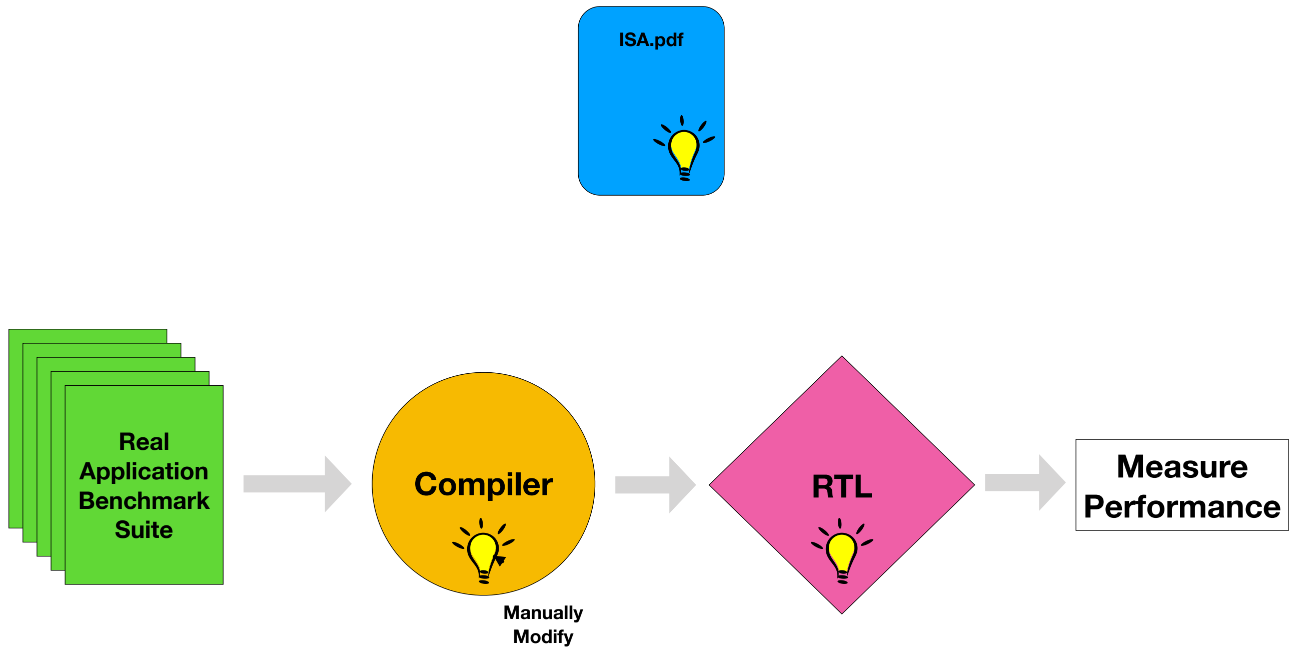
ISA.pdf

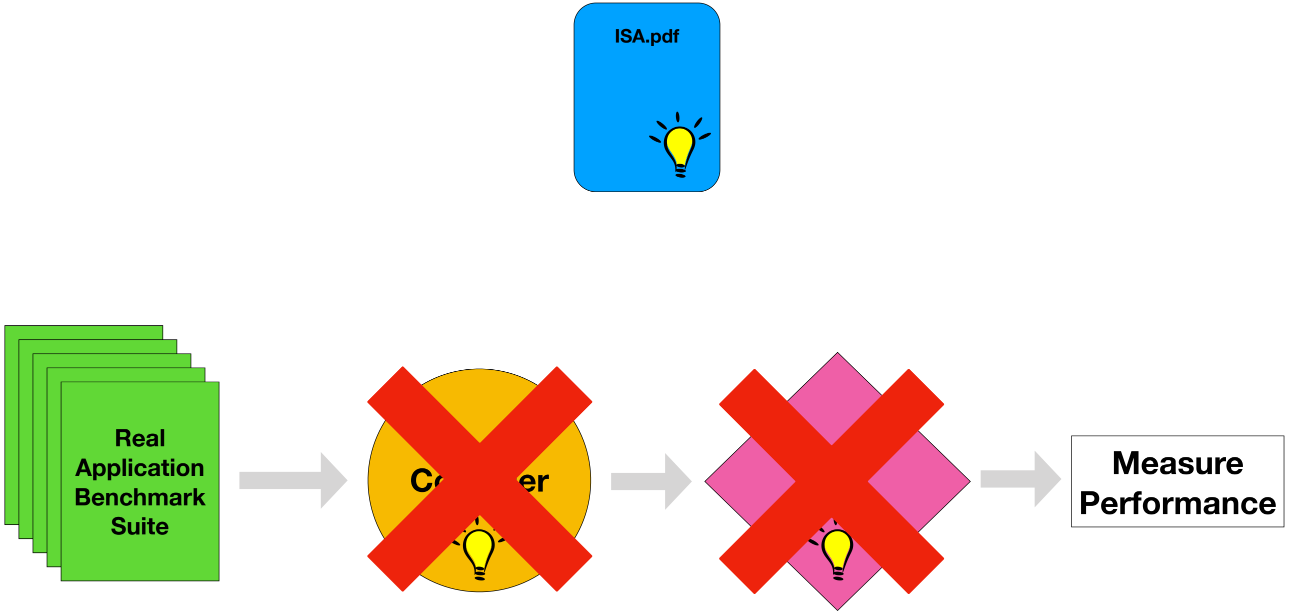

ISA.pdf
Modification Idea!


ISA.pdf
Real Application Benchmark Suite
Compiler
RTL
Measure Performance

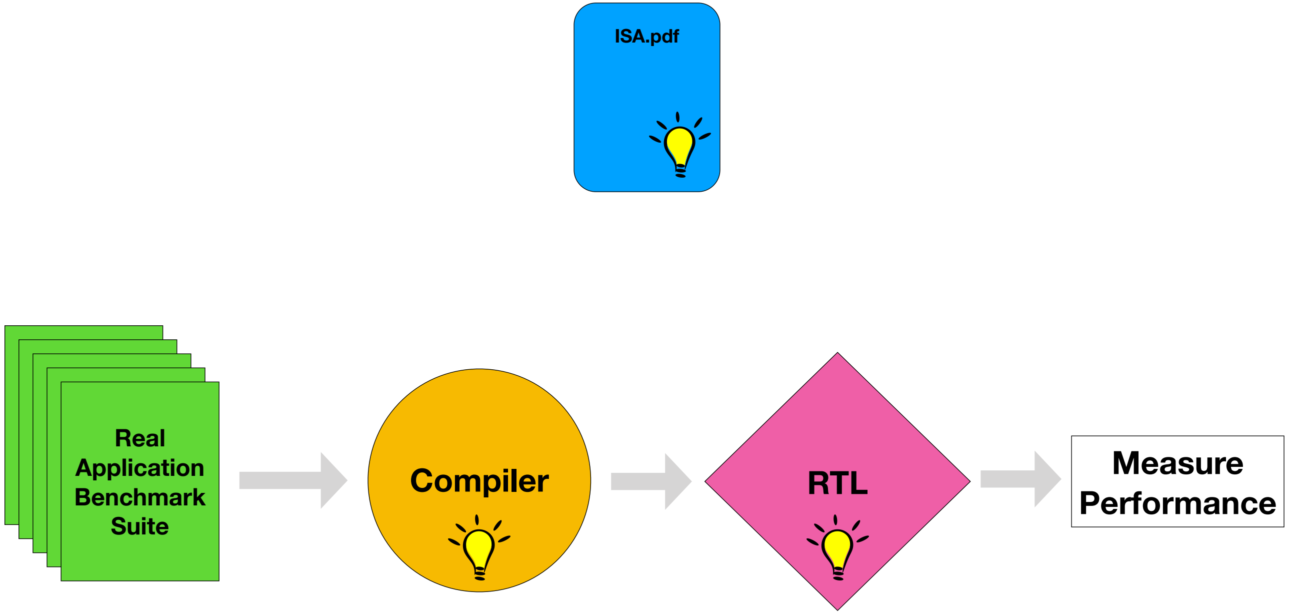

ISA.pdf
Real Application Benchmark Suite
Compiler
RTL
Manually Modify
Measure Performance


ISA.pdf
Real Application Benchmark Suite
Co er
Broken!
RTL
Measure Performance

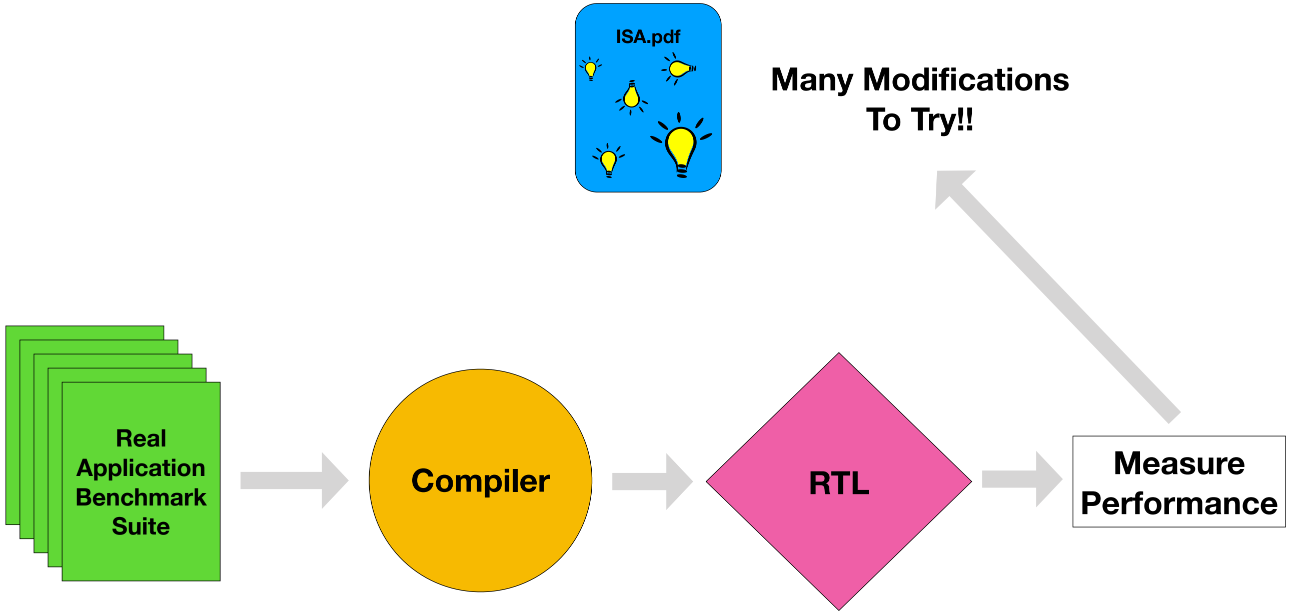

ISA.pdf
Real Application Benchmark Suite
Compiler
Manually Modify
RTL
Measure Performance

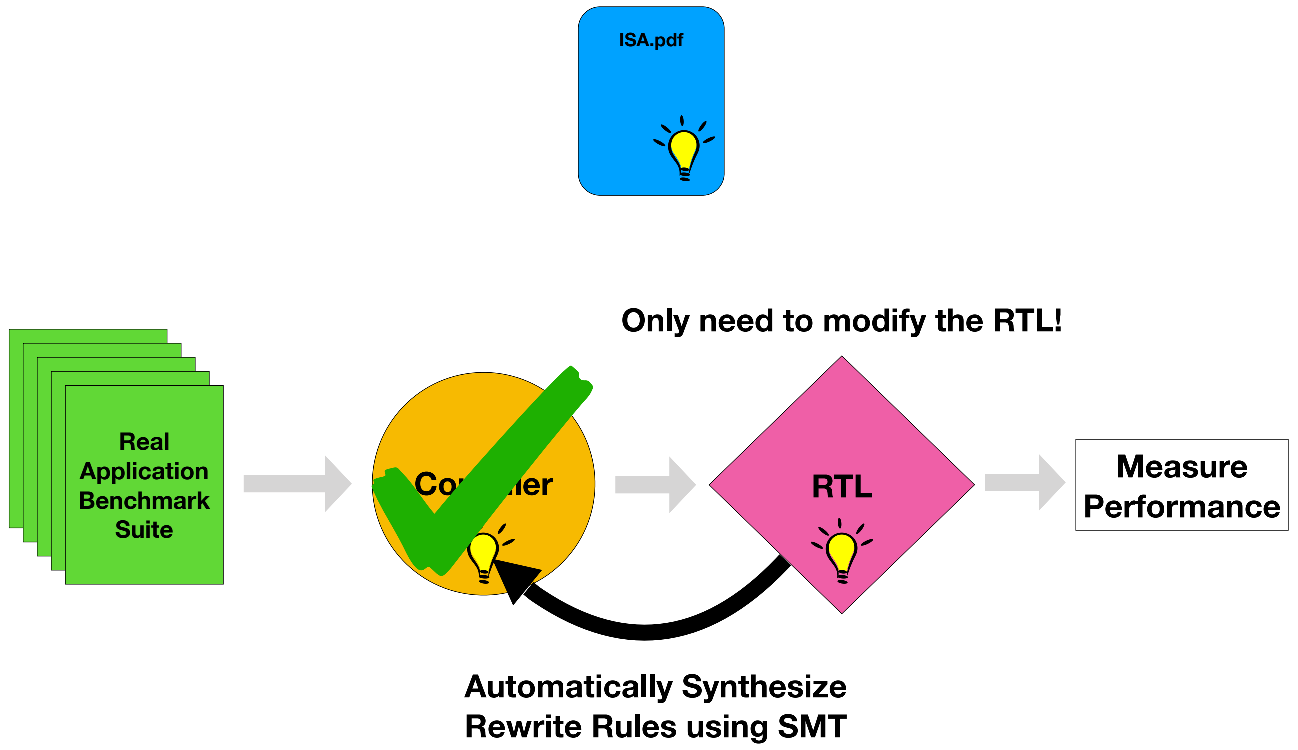


Probably Still Broken!
Need to test and debug



Finally can benchmark but that was time consuming and error prone!


ISA.pdf
Many Modifications To Try!!
Real Application Benchmark Suite
Compiler
RTL
Measure Performance


ISA.pdf
Only need to modify the RTL!
Real Application Benchmark Suite
Co ler
RTL
Measure Performance
Automatically Synthesize Rewrite Rules using SMT

Synthesizing Rewrite Rules for Diverse Architectures

Ross Daly, Caleb Donovick, Jack Melchert, Raj Setaluri, Nestan Tsiskaridze, Priyanka Raina, Clark Barrett, Pat Hanrahan

Olivia Hsu

Accelerating Sparse Tensor Algebra

Too many tensor kernels for fixed-function libraries and backends

$a = Bc$ Linear Algebra

$a = Bc + a$

$a = Bc + b$ $A = B + C$ $a = \alpha Bc + \beta a$

$a = B^T c$ $A = \alpha B$ $a = B(c + d)$

$a = B^T c + d$ $A = B + C + D$ $A = BC$

$A = B \odot C$ $a = b \odot c$ $A = 0$ $A = B \odot (CD)$

$A = BCd$ $A = B^T$ $a = B^T Bc$

$a = b + c$ $A = B$ $K = A^T CA$

$A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj}$ $A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij}$

$A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj}$ $A_{ij} = \sum_{k} B_{ijk} c_k$

$A_{ijk} = \sum_{l} B_{ikl} C_{lj}$ $A_{ik} = \sum_{j} B_{ijk} c_j$ Data analytics (tensor factorization)

$A_{jk} = \sum_{i} B_{ijk} c_i$ $A_{ijl} = \sum_{k} B_{ikl} C_{kj}$

$\tau = \sum_i z_i (\sum_j z_j \theta_{ij})(\sum_k z_k \theta_{ik})$

$C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}}\, \overline{P_{il}}$

$a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}}\, \overline{P_{ip}}$

Quantum Chromodynamics

×

Dense Matrix
CSR DCSR BCSR
COO ELLPACK CSB
Blocked COO CSC
DIA Blocked DIA DCSC
Sparse vector Hash Maps
Coordinates
CSF Dense Tensors
Blocked Tensors
Linked Lists Database
Compression Schemes
Cloud Storage

×

CPU
GPUs TPUs
FPGA CGRAs
Sparse Tensor Hardware
Cloud Computers
Supercomputers

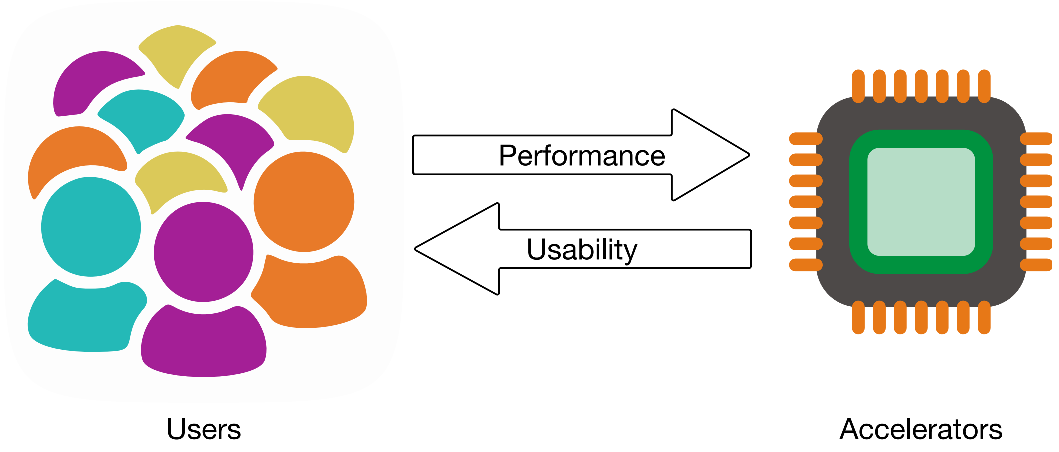
×

reorder
precompute
parallelize split
map divide
vectorize unroll
position

Olivia Hsu

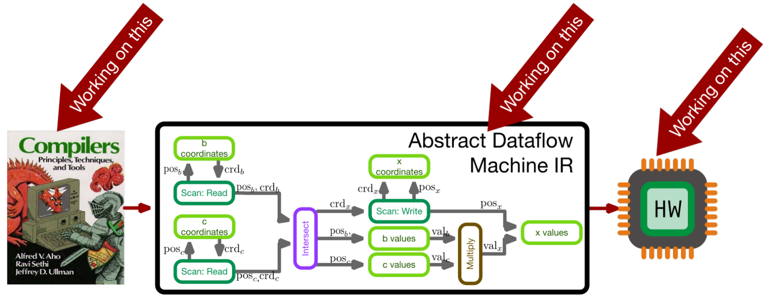
Accelerating Sparse Tensor Algebra

Long Tail of Expressions (algorithm) × Varying Compression Structures (format) × More Performant Backends (platform) × Backend-Specific Transformations (schedules)



Olivia Hsu

Accelerating Sparse Tensor Algebra

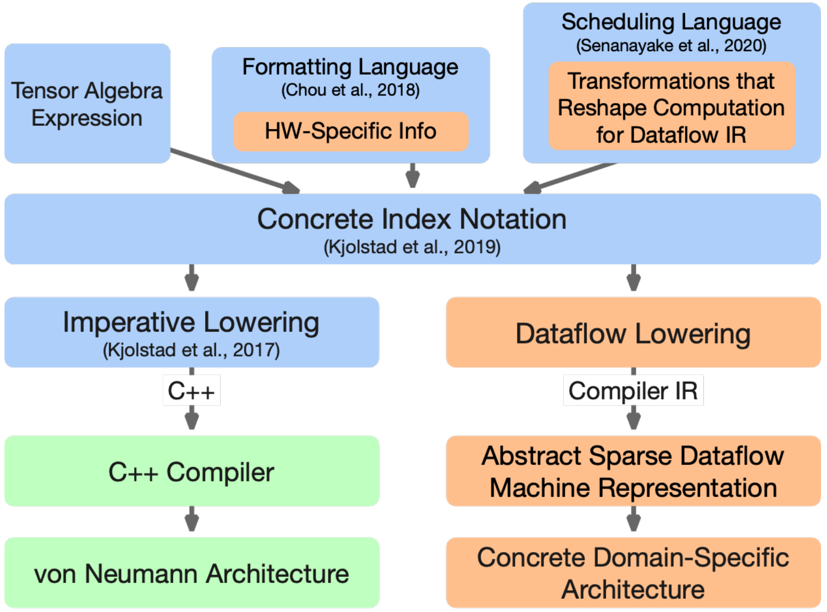


Legend

🟧 Sparse Dataflow Compiler

🟩 Other

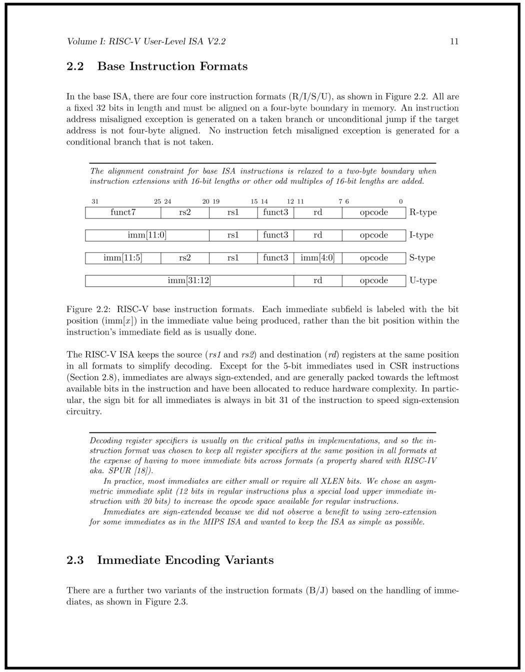
🟦 TACO Literature



PEak:
The Single Source of Truth

Caleb Donovick

State of the Art Specification



2.2 Base Instruction Formats

In the base ISA, there are four core instruction formats (R/I/S/U), as shown in Figure 2.2. All are a fixed 32 bits in length and must be aligned on a four-byte boundary in memory. An instruction address misaligned exception is generated on a taken branch or unconditional jump if the target address is not four-byte aligned. No instruction fetch misaligned exception is generated for a conditional branch that is not taken.

The alignment constraint for base ISA instructions is relaxed to a two-byte boundary when instruction extensions with 16-bit lengths or other odd multiples of 16-bit lengths are added.

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

Figure 2.2: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding. Except for the 5-bit immediates used in CSR instructions (Section 2.8), immediates are always sign-extended, and are generally packed towards the leftmost available bits in the instruction and have been allocated to reduce hardware complexity. In particular, the sign bit for all immediates is always in bit 31 of the instruction to speed sign-extension circuitry.

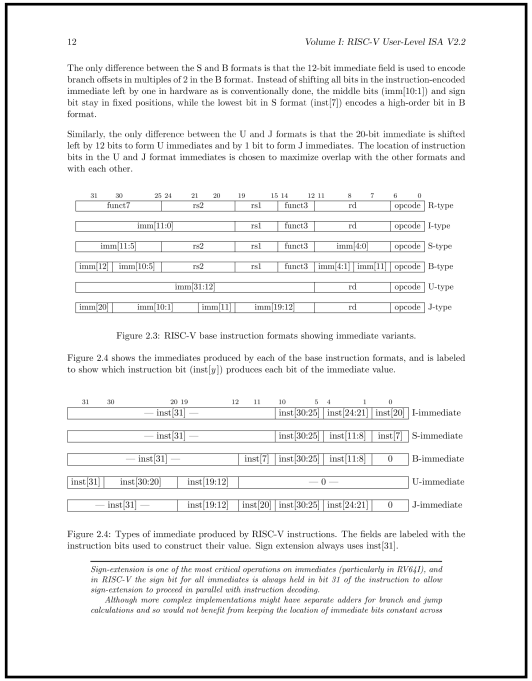
Decoding register specifiers is usually on the critical paths in implementations, and so the instruction format was chosen to keep all register specifiers at the same position in all formats at the expense of having to move immediate bits across formats (a property shared with RISC-IV aka. SPUR [18]).

In practice, most immediates are either small or require all XLEN bits. We chose an asymmetric immediate split (12 bits in regular instructions plus a special load upper immediate instruction with 20 bits) to increase the opcode space available for regular instructions.

Immediates are sign-extended because we did not observe a benefit to using zero-extension for some immediates as in the MIPS ISA and wanted to keep the ISA as simple as possible.

2.3 Immediate Encoding Variants

There are a further two variants of the instruction formats (B/J) based on the handling of immediates, as shown in Figure 2.3.



The only difference between the S and B formats is that the 12-bit immediate field is used to encode branch offsets in multiples of 2 in the B format. Instead of shifting all bits in the instruction-encoded immediate left by one in hardware as is conventionally done, the middle bits (imm[10:1]) and sign bit stay in fixed positions, while the lowest bit in S format (inst[7]) encodes a high-order bit in B format.

Similarly, the only difference between the U and J formats is that the 20-bit immediate is shifted left by 12 bits to form U immediates and by 1 bit to form J immediates. The location of instruction bits in the U and J format immediates is chosen to maximize overlap with the other formats and with each other.

31	30–25	24–21	20	19–15	14–12	11–8	7	6–0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

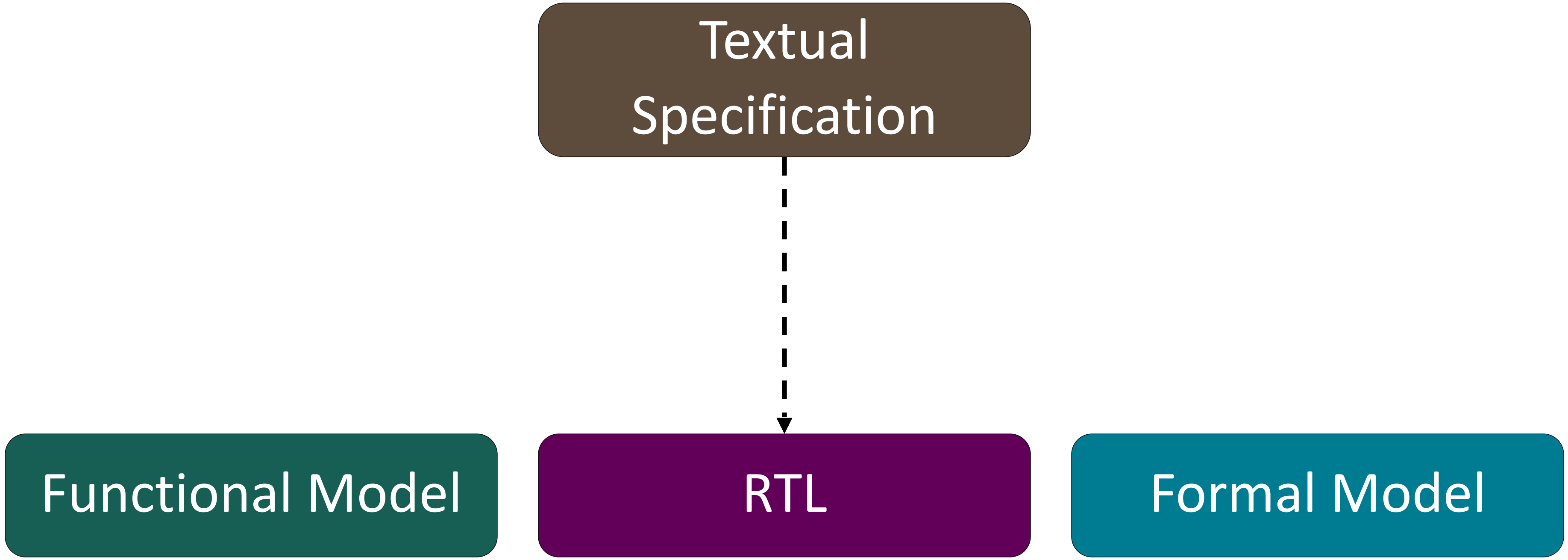
31	30–20	19–12	11	10–5	4–1	0	
— inst[31] —				inst[30:25]	inst[24:21]	inst[20]	I-immediate
— inst[31] —				inst[30:25]	inst[11:8]	inst[7]	S-immediate
— inst[31] —			inst[7]	inst[30:25]	inst[11:8]	0	B-immediate
inst[31]	inst[30:20]	inst[19:12]	— 0 —				U-immediate
— inst[31] —		inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate

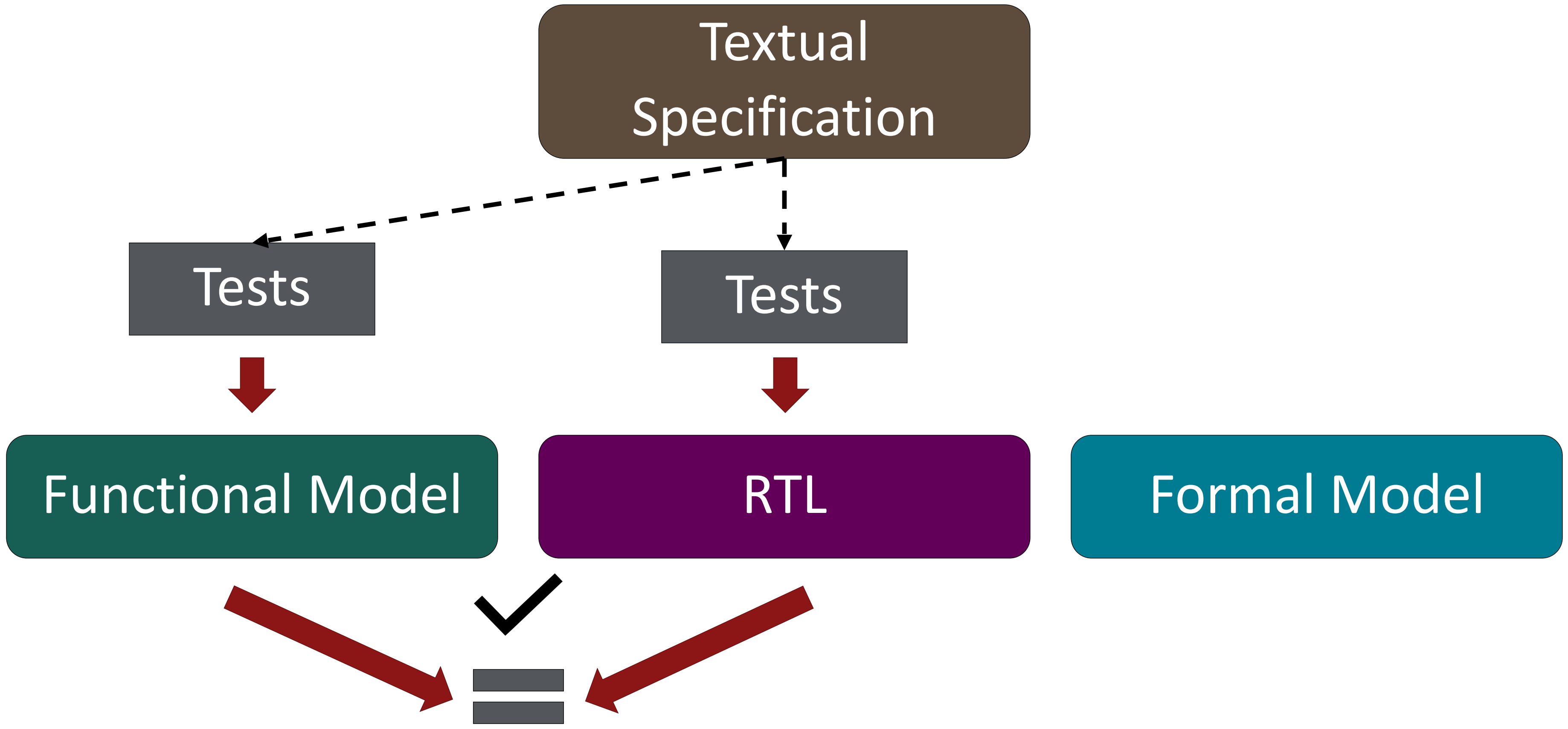
Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

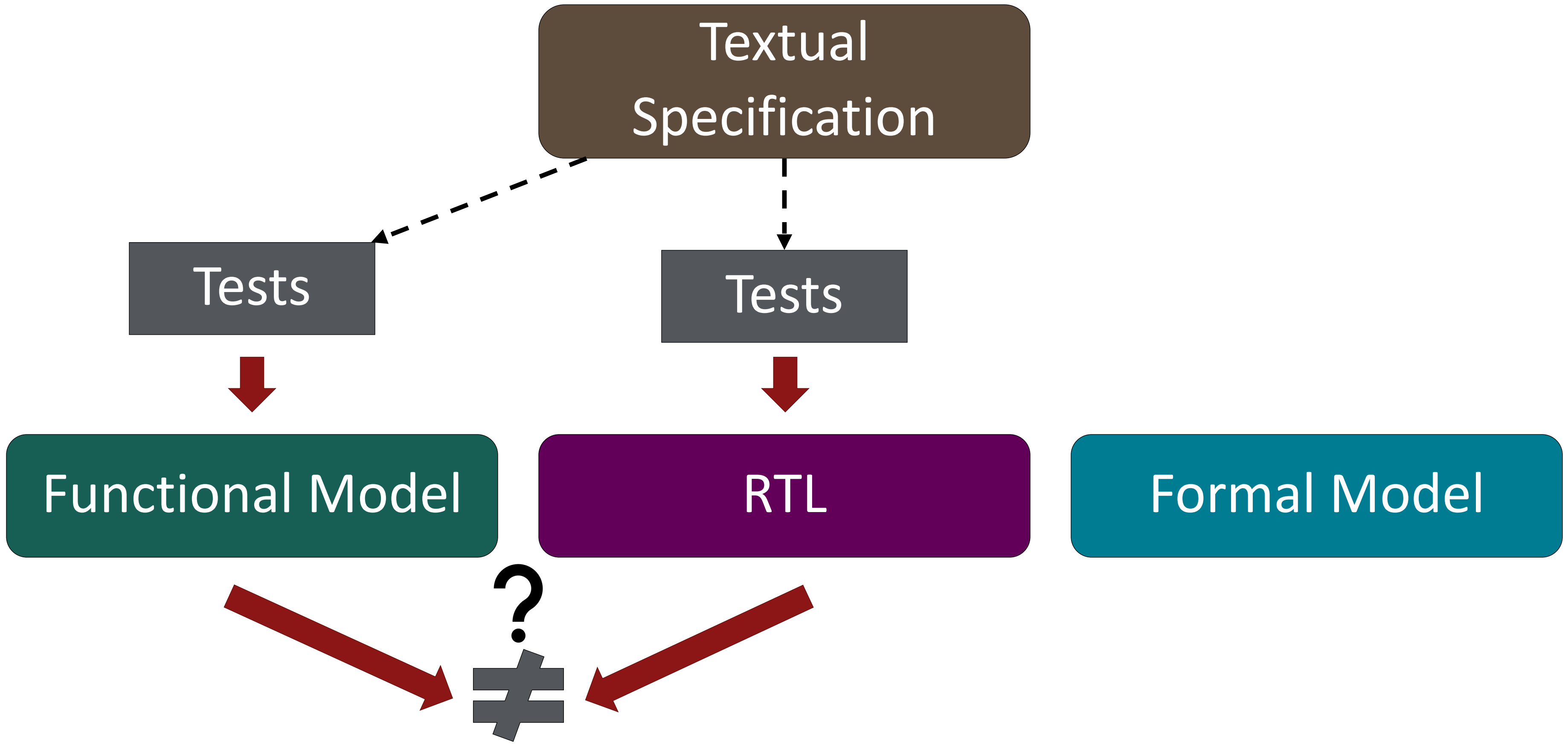
Sign-extension is one of the most critical operations on immediates (particularly in RV64I), and in RISC-V the sign bit for all immediates is always held in bit 31 of the instruction to allow sign-extension to proceed in parallel with instruction decoding.

Although more complex implementations might have separate adders for branch and jump calculations and so would not benefit from keeping the location of immediate bits constant across

The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2. RISC-V Foundation, 2017

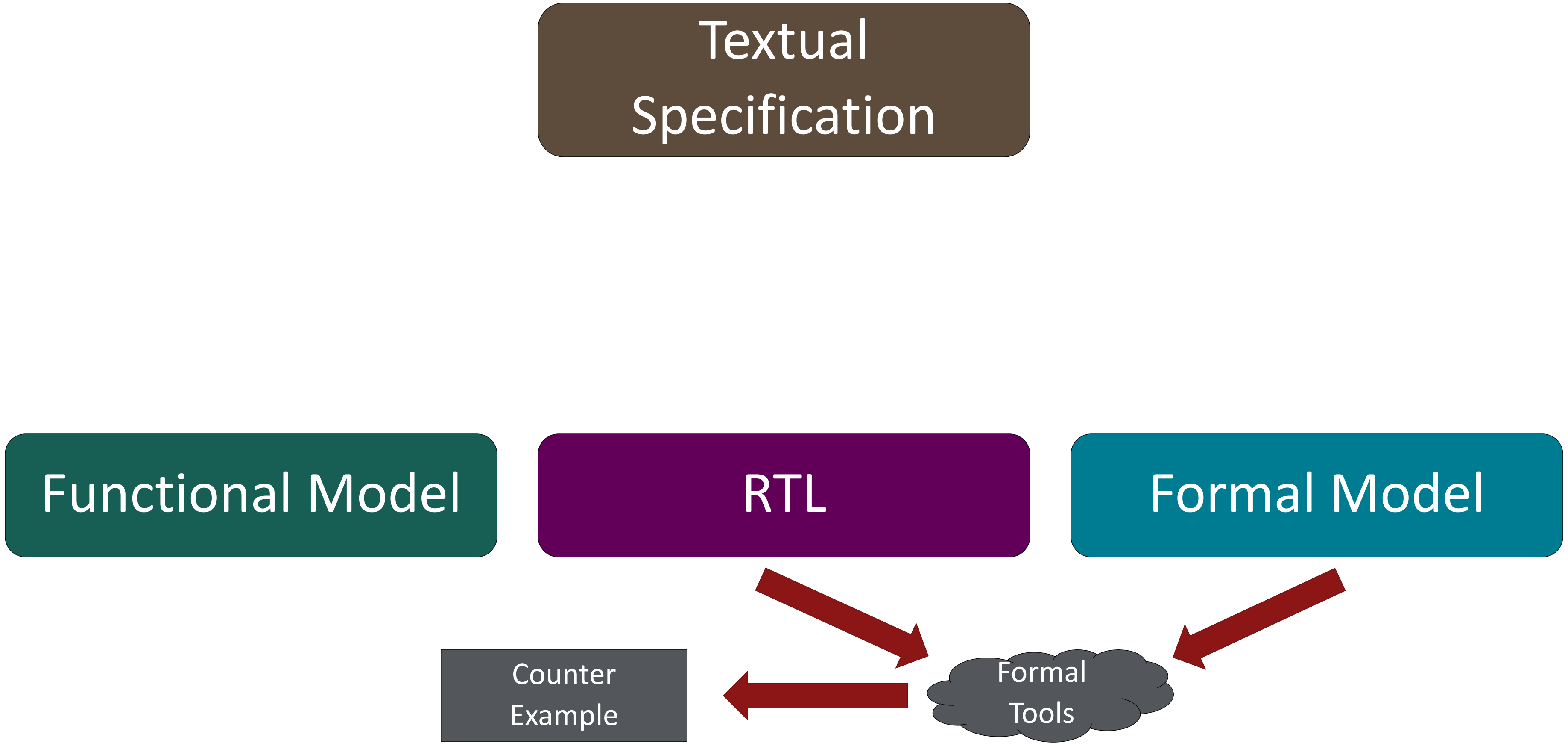

Textual Specification
Functional Model
RTL
Formal Model


Textual Specification
Tests
Tests
Functional Model
RTL
Formal Model

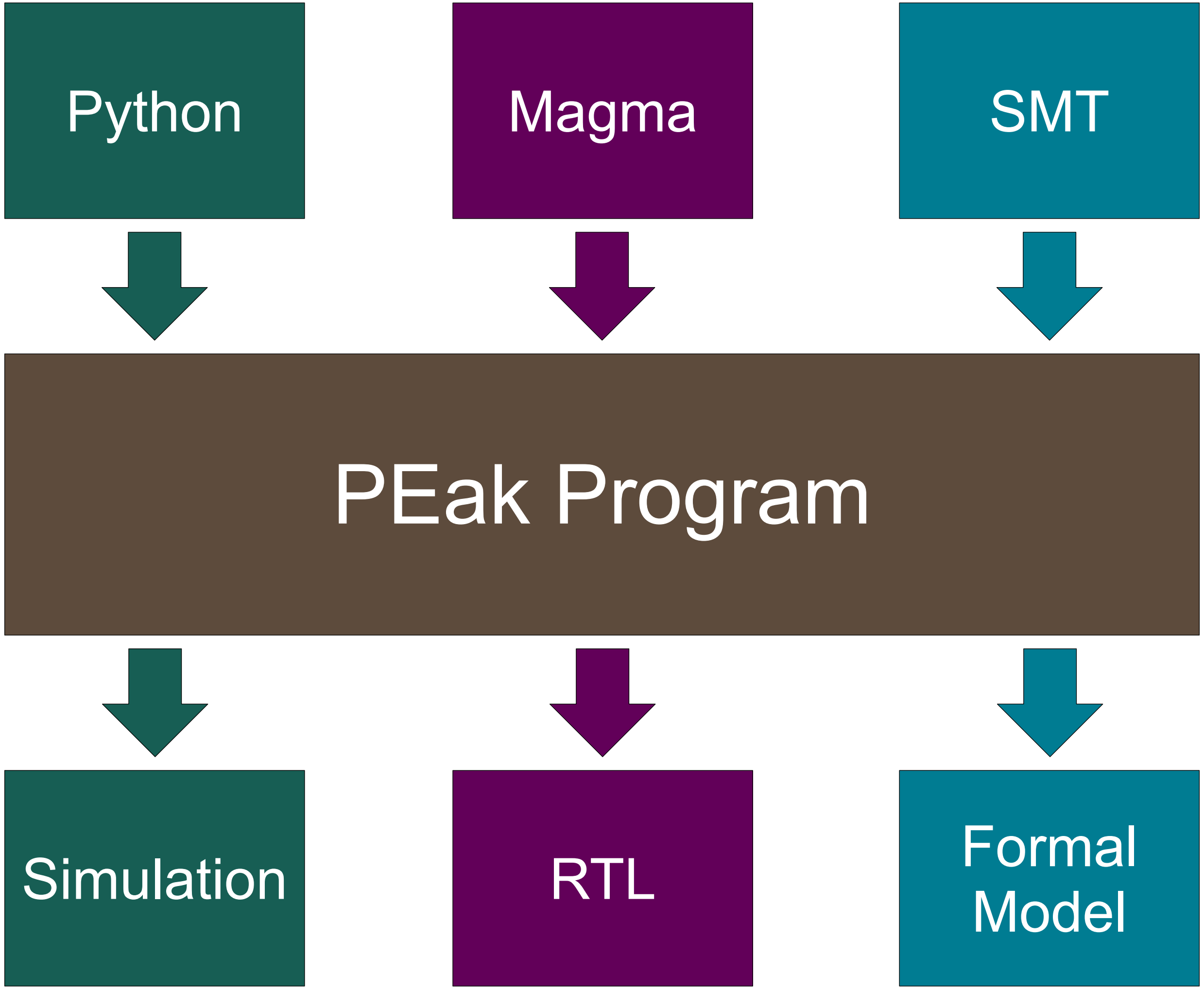

Textual Specification
Tests
Tests
Functional Model
RTL
Formal Model
?
≠

What went wrong?

- Functional Model might be wrong
- RTL might be wrong
- Might be unspecified behavior
- Tests might differ


Textual Specification
Functional Model
RTL
Formal Model
Formal Tools
Counter Example

No way to test textual specification


Python
Magma
SMT
PEak Program
Simulation
RTL
Formal Model

PEak has many features

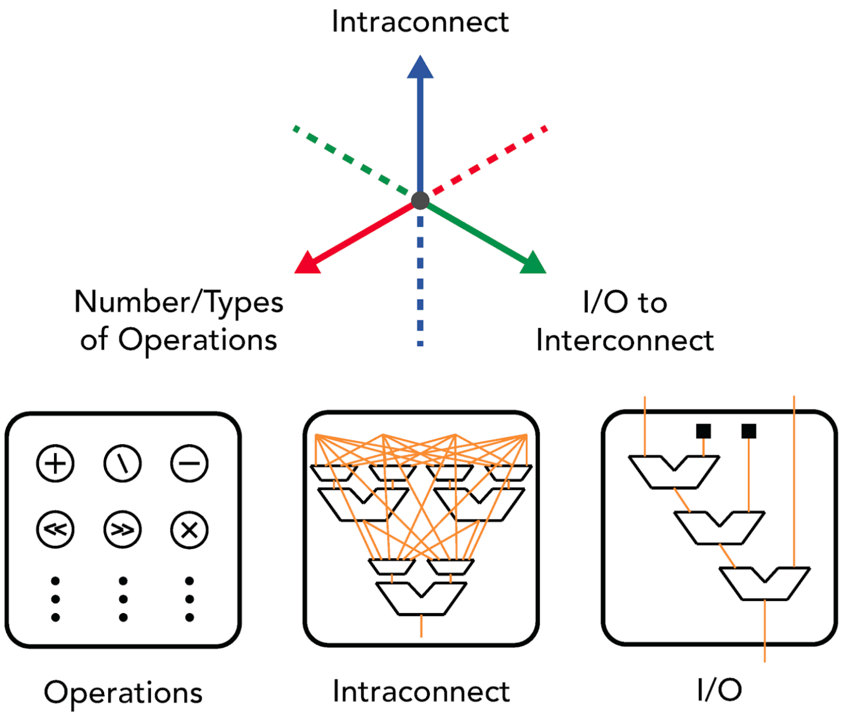
See my poster

Automated Design Space Exploration of CGRA Processing Element Architectures using Frequent Subgraph Analysis

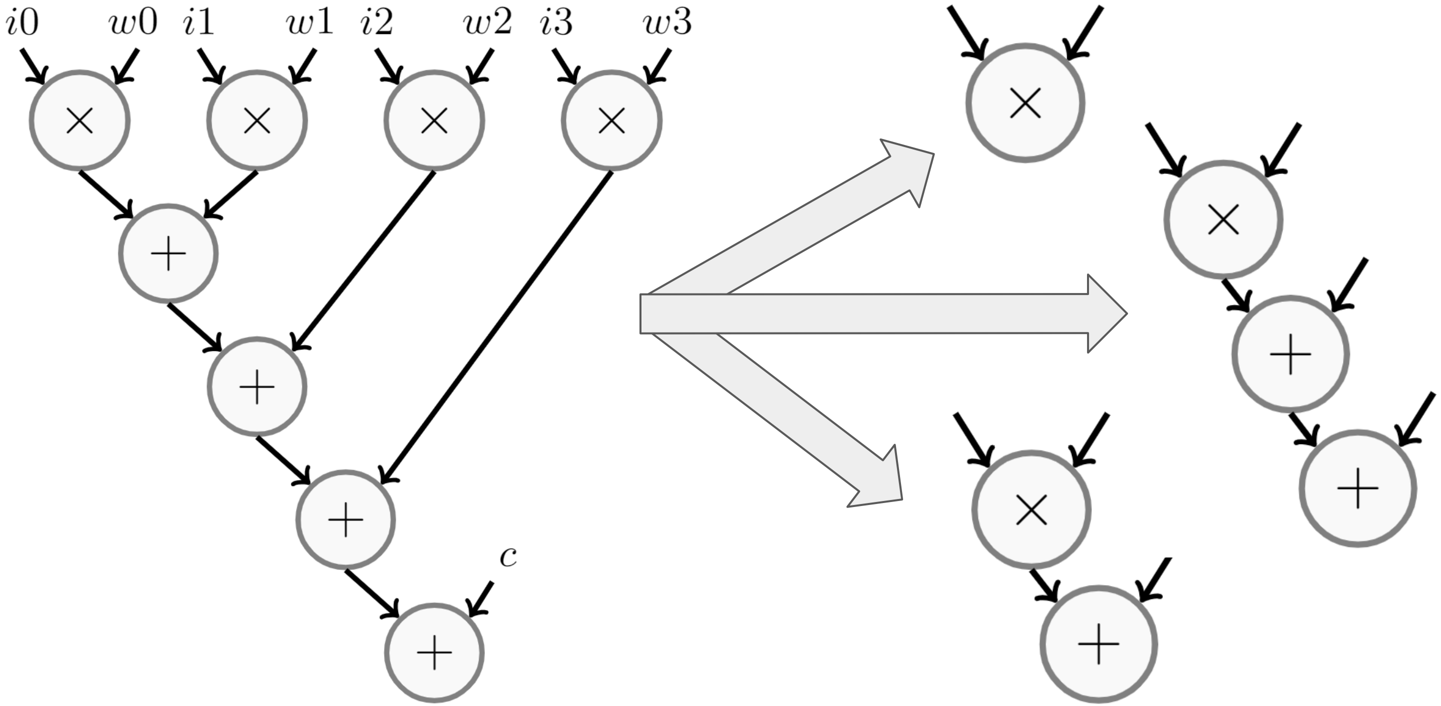
Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly

How can we generate an optimal PE architecture?

1. Analyze application domain benchmarks to find possible optimizations
2. Quickly create PE designs that explore the design space
3. Automatically generate full compiler to run applications

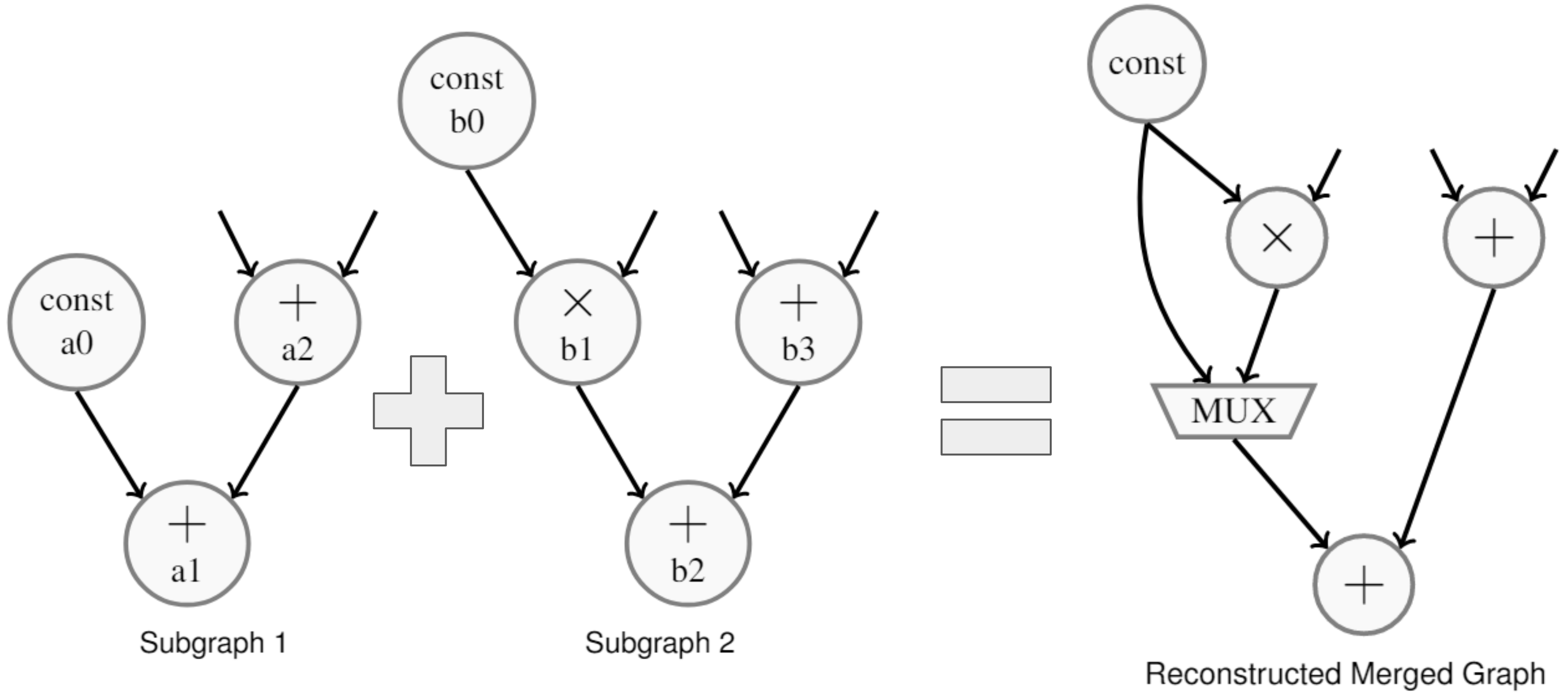


Application Analysis - Frequent Subgraph Mining



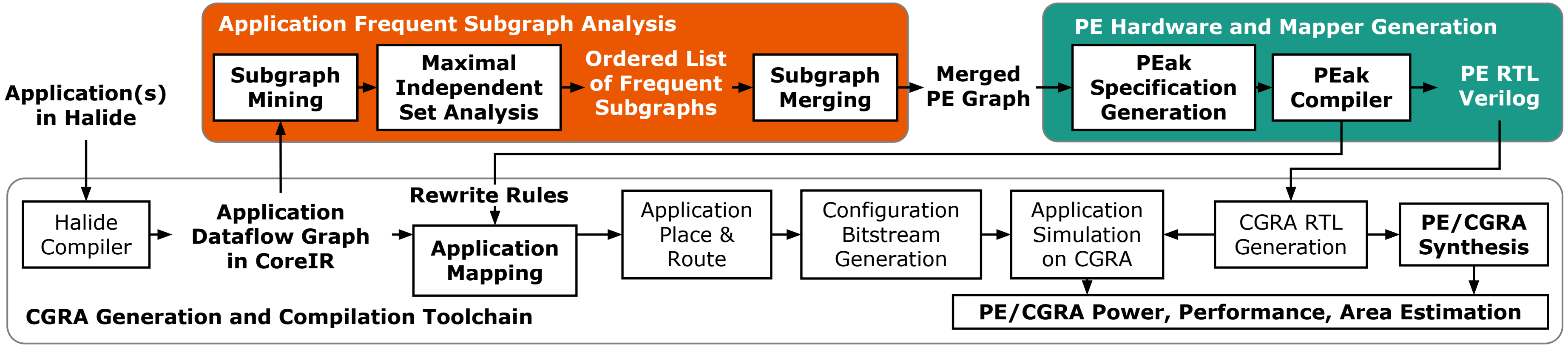
Frequent subgraphs represent common computational blocks

Producing PEs - Frequent Subgraph Merging



Merging frequent subgraphs results in efficient and performant PEs

Design Space Exploration Framework



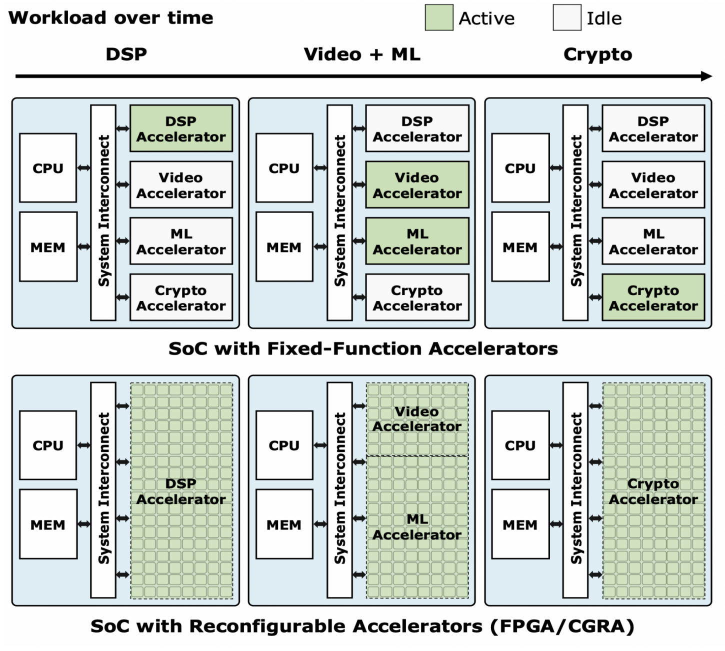
Application Frequent Subgraph Analysis
Subgraph Mining
Maximal Independent Set Analysis
Ordered List of Frequent Subgraphs
Subgraph Merging
Merged PE Graph
PE Hardware and Mapper Generation
PEak Specification Generation
PEak Compiler
PE RTL Verilog
Application(s) in Halide
Rewrite Rules
Halide Compiler
Application Dataflow Graph in CoreIR
Application Mapping
Application Place & Route
Configuration Bitstream Generation
Application Simulation on CGRA
CGRA RTL Generation
PE/CGRA Synthesis
PE/CGRA Power, Performance, Area Estimation
CGRA Generation and Compilation Toolchain

Dynamic Partial Reconfiguration

Kalhan Koul - Rising 3rd Year PhD

Motivation for Dynamic Partial Reconfiguration

- Definition: Reconfigure parts of the CGRA (***partial***) without affecting other parts at runtime (***dynamic***)
- Example: fixed-function accelerators vs reconfigurable accelerators



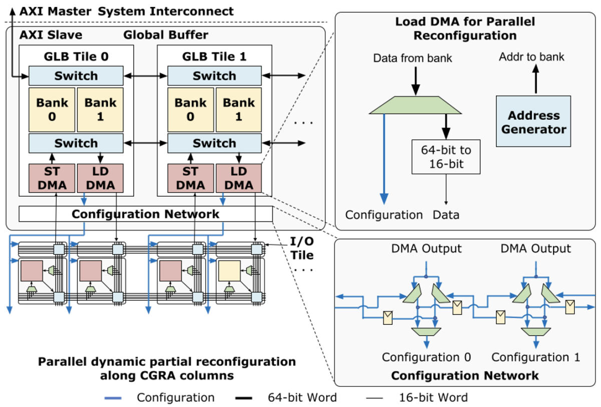
DSP + Video + ML + Crypto
- + Highly optimized accelerators
- - Cannot add new accelerators
- - Low hardware utilization

DSP->*(Reconf.)*->Video+ML->*(Reconf.)*->Crypto
- + Flexible to run any accelerator
- + High hardware utilization
- - Reconfigurable hardware overhead
- - Reconfiguration time overhead

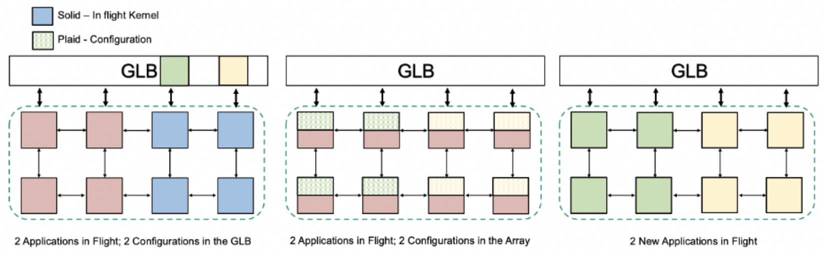
Architectural Exploration

- Goal: develop and quantify the benefits of hardware architectural additions on top of a baseline CGRA, including : (a) Relocatable bitstream (b) Partial Reconfiguration (PR) region shape and interconnect network (c) Parallel Reconfiguration (d) Double Buffer Reconfiguration
- Visit my poster to see the implementation/exploration of each!

Ex. Parallel Reconfiguration

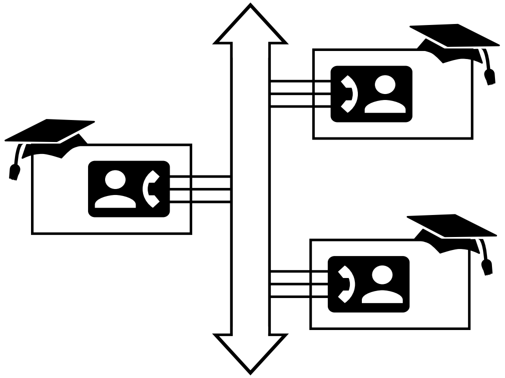


Ex. Double Buffer Reconfiguration


Solid – In flight Kernel
Plaid - Configuration
GLB
2 Applications in Flight; 2 Configurations in the GLB
2 Applications in Flight; 2 Configurations in the Array
2 New Applications in Flight

SMART COMPONENTS

AVOIDING LATE STAGE DESIGN BUGS USING SESSION TYPES



Lenny Truong

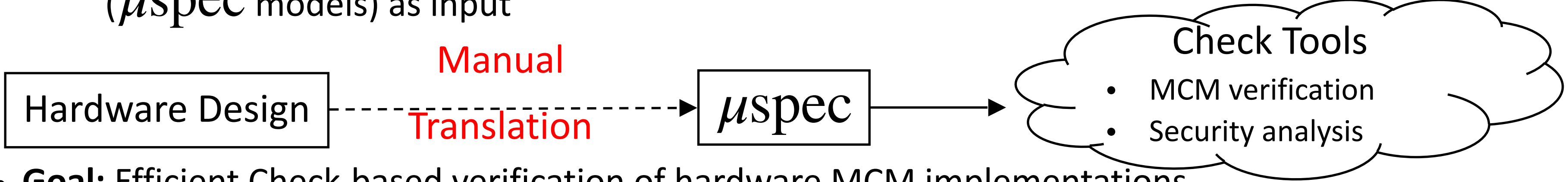
- Lack of abstraction and imprecise specifications leads to debugging using gate level simulations
- **Smart Components** surfaces these bugs in RTL
  - **Abstract actions** capture component interfaces
  - **Session types** verify composition of components
  - **Unit testing** verifies concrete implementations

lenny@cs.stanford.edu

Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations

Yao Hsiao, Dominic P. Mulligan*, Nikos Nikoleris*, Gustavo Petri*, Caroline Trippel
Stanford University, *ARM Research

- **Motivation:**
  - Memory Consistency Models (MCM) specify legal outcomes of shared memory programs in multiprocessor
  - Check tools[1] requires manually-constructed formal microarchitecture specifications (μspec models) as input

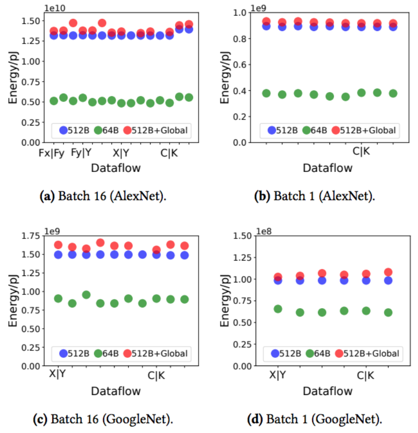


- **Goal:** Efficient Check-based verification of hardware MCM implementations
- **Key Challenges:**
  - Decomposition of complete μspec models
  - Gap between <u>operational</u> RTL and <u>axiomatic</u> μspec models

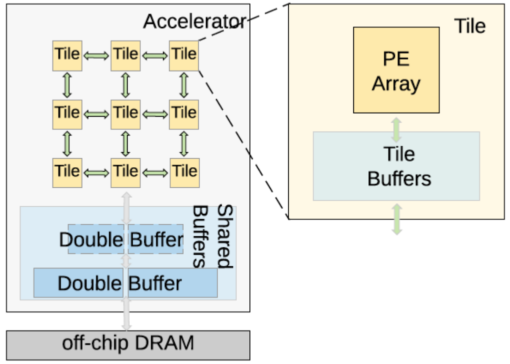
Improving Energy Efficiency for DNNs on CGRAs with Local Storage in the PEs

Ankita Nayak

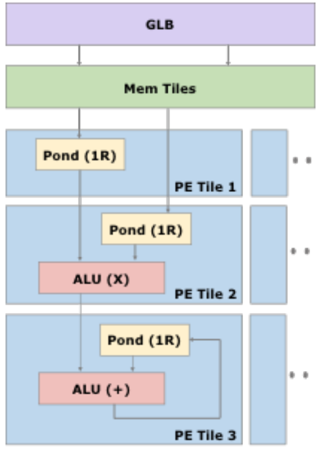
- With proper blocking schemes many dataflows can achieve close-to-optimal energy efficiency
- Memory resource allocation has a larger impact on DNN energy
- **Goal:** Introduce a new low-access-cost memory hierarchy (Ponds) to improve energy efficiency
- **Challenge:** Should be extremely area and energy efficient, yet flexible enough to map different energy efficient schedules



Design space of dataflow with optimal loop blocking schemes



DNN Accelerator Template with Hierarchical Memories



Introducing Ponds in Amber CGRA

Keyi Zhang

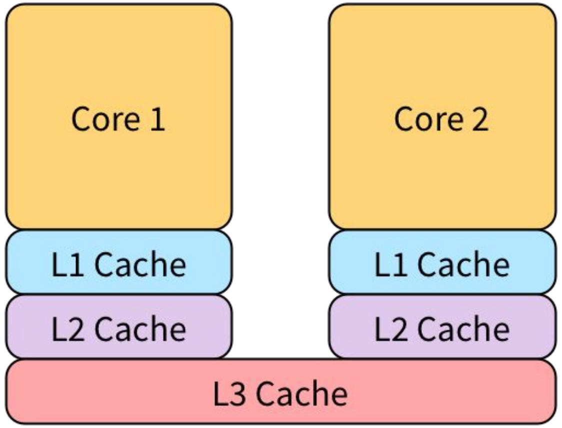
System-Level SoC Verification Framework

Thread 1

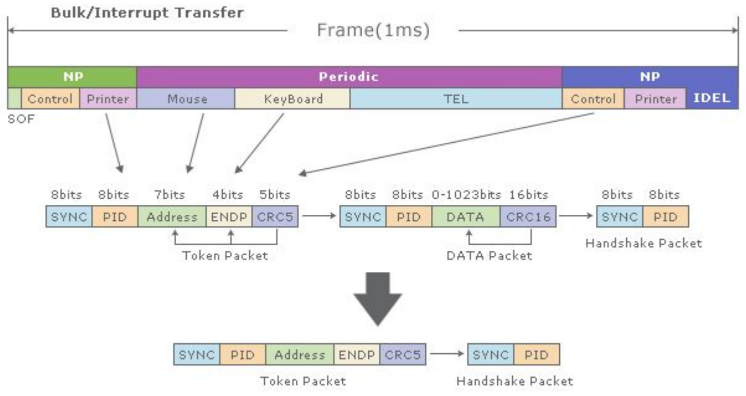
```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```



Memory consistency bug?



Interrupt not received properly?

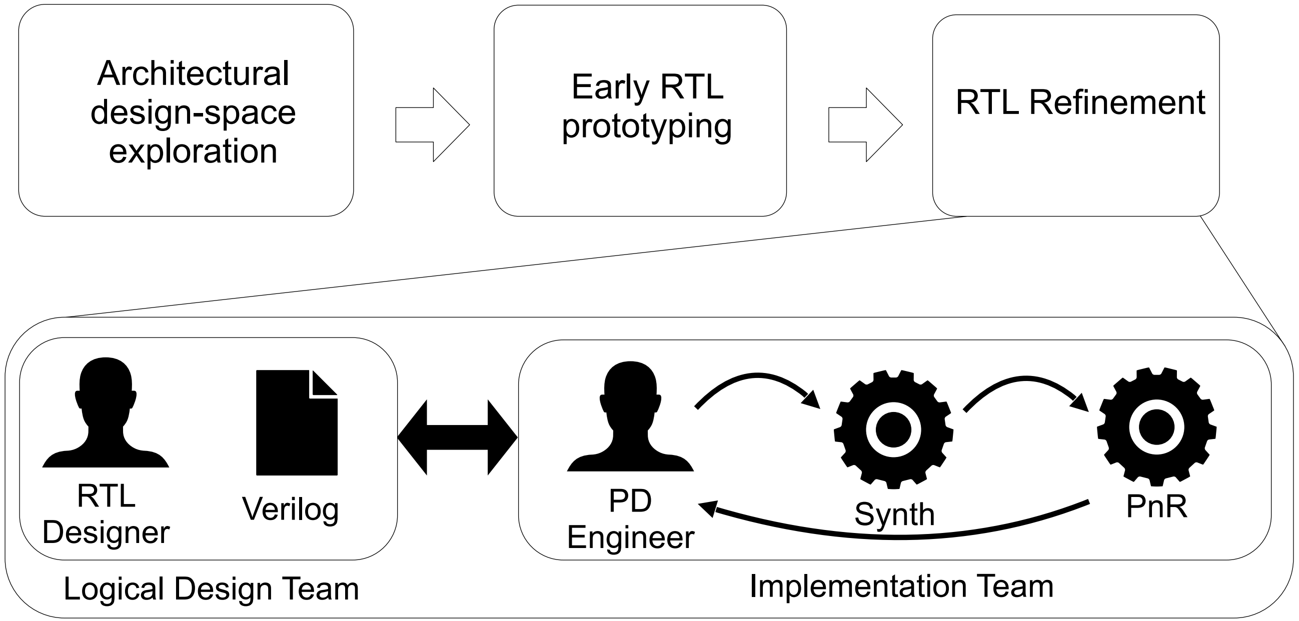
Is there a bug?HW or SW?

Utilizing Hardware Generators for Agile RTL Refinement

Raj Setaluri, Alex Carsello, James Thomas, and Christopher Torng

Stanford University

RTL refinement is a slow, iterative process



- Refinement is a cross-team process
- Bogged down by tool spin-time
- Have to up-level reports to source

A Source-Level Development Platform for Agile RTL Refinement

- Intelligently slice the circuit to get the parts you care about
- Query against source-level names
- Abstract away tool complexity

```
u_over_4 = u >> 2
u_over_2 = u >> 1
u_over_2_plus_B_over_2 = csa(u_over_2, B) >> 1
u_plus_B_over_4 = csa(u, B) >> 2
u_plus_B_over_2 = csa(u, B) >> 4
u_plus_B_over_2_B_over_2 = csa(u_plus_B_over_2, B) >> 1
odd_update_0 = odd_update(csa(u, y), B)
odd_update_1 = odd_update(csa(u, y), B)
```

```
>>> report_timing(from_=[u, B], to=[u_plus_B_over_2_B_over_2])
{
  "B[0]":                      0.01,
  "add_inst2.in1[0]":          0.01,
  "add_inst2.out[15]":         1.12,
  "lshr_inst4.in0[15]":        1.12,
  "lshr_inst4.out[9]":         1.21,
  "add_inst3.in0[9]":          1.21,
  "add_inst3.out[14]":         1.57,
}
```

A General-Purpose Memory System for Data-Intensive Accelerators

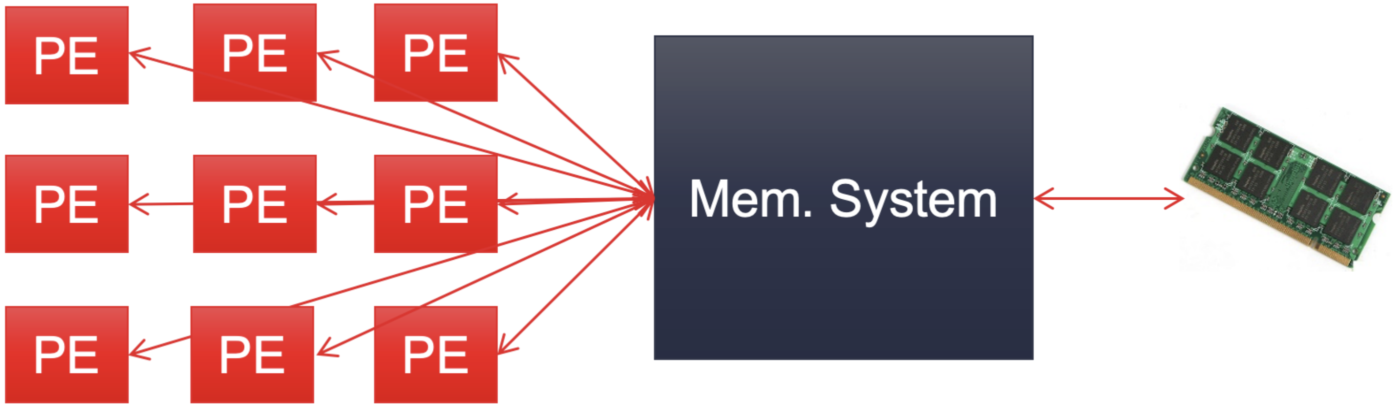
James Thomas

Data-Intensive Accelerator Design Platform Required

- Designing complex accelerator from scratch is way too expensive
- CUDA programming is relatively easy -- you write code for one thread and then it is run in parallel on a huge number of cores to get high performance
    - Can we have a similar model for accelerator design?

CUDA-like Accelerator Design Model

- Design and verify a single processing element (PE) that communicates in an AXI-like protocol to memory
- Platform replicates this PE (100x+) into memory access fabric that handles communication with DRAM



Toy CGRA:

Evaluating the Technology Portability of Agile Hardware Design Flow

PO-HAN CHEN

Overview

- Toy CGRA is a course project of EE272B

Stanford AHA!
Tool Chain

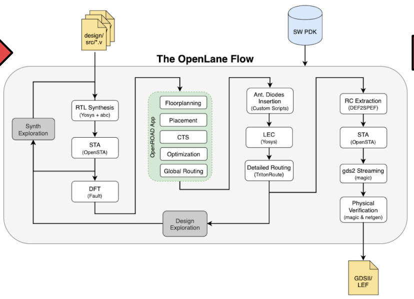
HW Generation

App Scheduling

App/HW Mapping

OpenROAD

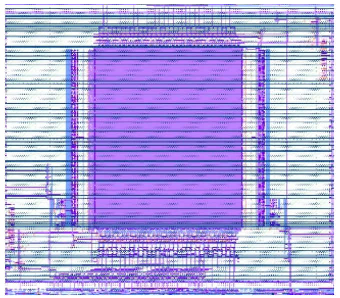
Digital OpenLane flow



Baskin Engineering UC SANTA CRUZ

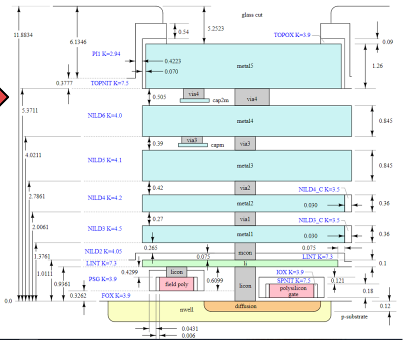
OpenRAM

SRAM Macro

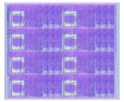


skywater

130nm Technology



EE 272B fast tape-out in 3 months!



Copy-and-Patch Compilation

Haoran Xu and Fredrik Kjolstad

Stanford University

The Need For Fast Compilation

* JIT compilers: compilation at runtime.
  - database engine: SQL query $\rightarrow$ machine code
  - web browser: WebAssembly module $\rightarrow$ machine code
* Need to compile fast **AND** generate good code!
* Our solution: Copy-and-Patch.
* Provides extremely fast compilation AND decent generated code.

The Need For Fast Compilation

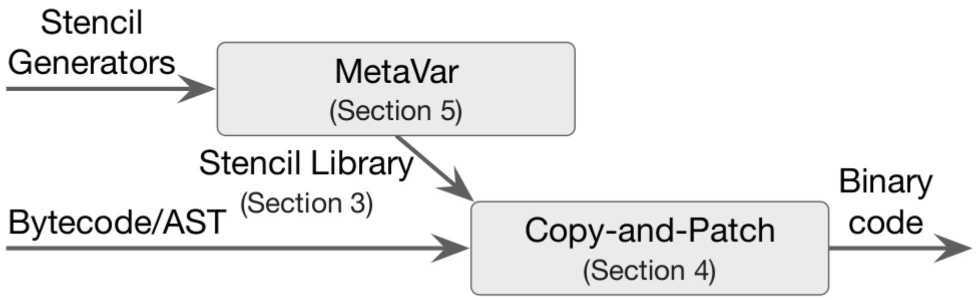
* JIT compilers: compilation at runtime.
  - database engine: SQL query $\rightarrow$ machine code
  - web browser: WebAssembly module $\rightarrow$ machine code
* Need to compile fast **AND** generate good code!
* Our solution: Copy-and-Patch.
* Provides extremely fast compilation AND decent generated code.

Copy-and-Patch Compilation

* Two example use cases: SQL compiler, WebAssembly compiler.
* Significantly outperforms existing approaches for fast compilation:
  - LLVM -O0 (100x faster compilation, 15% better code)
  - State-of-the-art baseline compilers from Chrome and Wasmer (5-20x faster compilation, 50%–60% better code)
  - Interpreters (10x faster execution)
* Works for both high-level languages (C-like) and low-level bytecodes (WebAssembly-like).

How it works

* No free lunch.
* But we can stand on the shoulders of giants.
* Cleverly using Clang+LLVM as a **preprocessor**.
* Pre-compute *a lot*, so little work to do at runtime.
* For more details: poster!



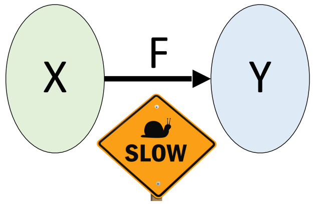
Fast Extended GCD for Large Integers for Verifiable Delay Functions

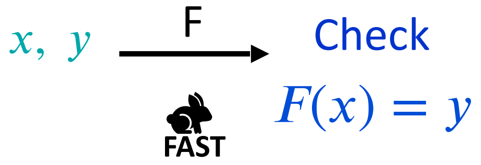
Kavya Sreedhar, Mark Horowitz, **Christopher Torng**

Verifiable delay function (VDF)

Allows one party (prover) to convince the other party (verifier) that a certain amount of time has passed

Mathematical Puzzle

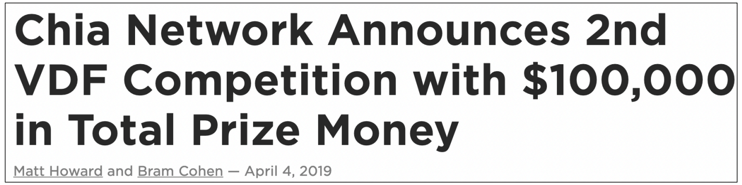




Delay: Inherently sequential work that is slow to compute

Verifiable: The output of the puzzle is easy to verify to be correct

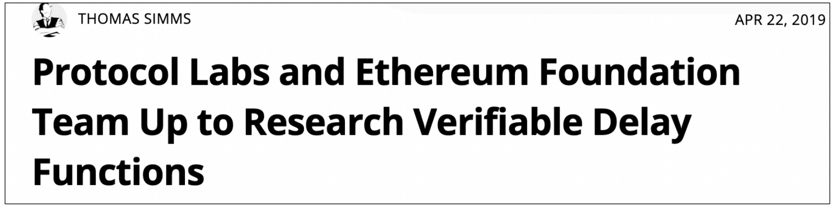
The crypto community is excited about VDFs



Chia Network Announces 2nd VDF Competition with $100,000 in Total Prize Money

Matt Howard and Bram Cohen — April 4, 2019

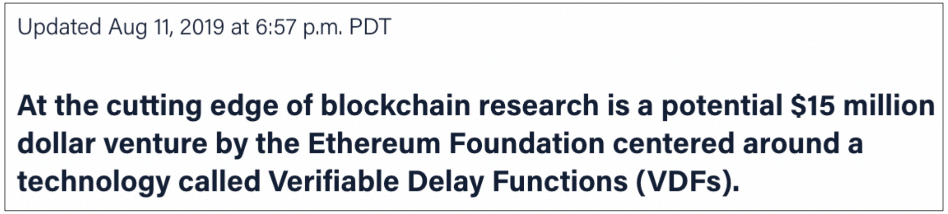
chia



THOMAS SIMMS APR 22, 2019

Protocol Labs and Ethereum Foundation Team Up to Research Verifiable Delay Functions

Protocol Labs



Updated Aug 11, 2019 at 6:57 p.m. PDT

At the cutting edge of blockchain research is a potential $15 million dollar venture by the Ethereum Foundation centered around a technology called Verifiable Delay Functions (VDFs).

ethereum foundation

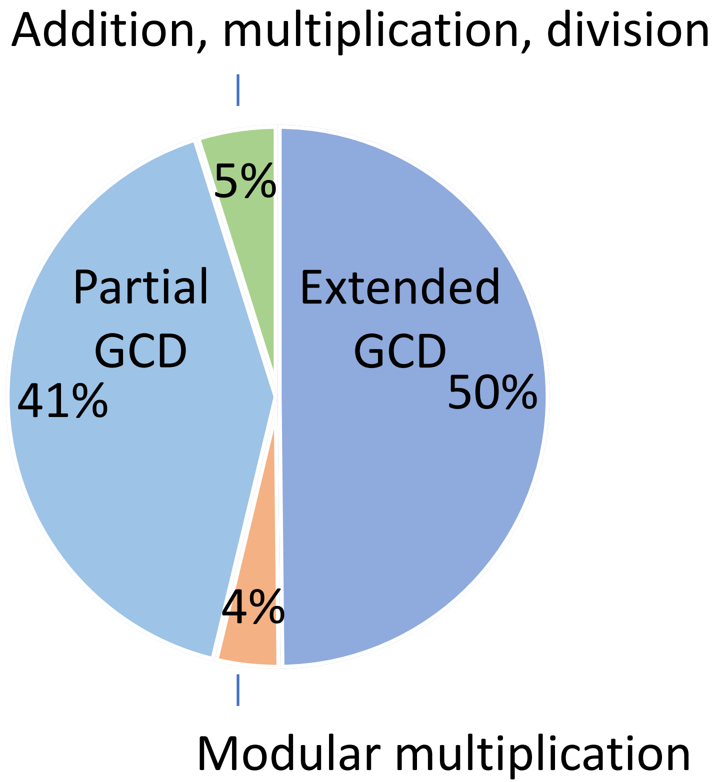
The speed of VDF evaluation directly impacts the security of these blockchains

Accelerating VDFs depends on fast extended GCD computations

Extended GCDs are cool again!

Come hear about arithmetic circuit optimizations in the context of fast VDFs

Come to our poster!


Addition, multiplication, division
Partial GCD
Extended GCD
Modular multiplication