Effective Formal Solutions to CPU Verification Challenges

Vikram Khosa

June 3rd, 2019
56th DAC, Las Vegas

arm

Outline

Overview

- **CPU Verification Challenges**
- **Application Models**
- **Focused Formal Application Examples**
  - Floating-Point Data-Path
  - Trace-Cache Macro-op Codec
  - Lock-Step CPU-level Determinism
  - Vector First-Fault Register Updates
- **Case-Study**
- **Conclusions**

Case-Study

- **Verifying System Register Accesses**
  - Definitions
  - Functionality
  - Why Formal?
  - DUT Interfaces
  - Ordering Rules
  - Interface Modeling
  - End-to-End Checkers
  - Complexity Analysis
  - Results and Execution
  - Example Bug Found

Overview

CPU Verification Challenges

- **Combinatorial Explosion**
  - Floating-Point Operations
  - Instruction Decode
  - Exception/Fault Reporting Logic
- **Ordering and Concurrency**
  - Out-of-Order (Memory/Register) Accesses
  - Cache-Coherence
- **Determinism**
  - Lockstep Execution

Application Models

Focused Formal

- **Opportunistic but Exhaustive**
  - Higher Locality
    - Smaller DUT Size
  - Narrower Scope
    - Specialized Problem
- **High RoI**
  - High Functional Complexity
  - Tractable Formal Complexity
  - Hard to Achieve Sufficient Coverage via Simulation

Comprehensive Formal

- **Global Properties**
  - Broader Problem/State Space
  - High Formal Complexity
- **Options**
  - **Non-Exhaustive**
    - Lower-Effort Bug-Hunting
    - Complementary to Simulation
  - **(Near-)Exhaustive**
    - Higher-Effort Complexity Management
    - Via Decomposition or Abstractions

Focused Formal

Examples

I. Floating-Point Datapath

Overview

- **Starting Point**
  - RTL vs. RTL equivalence against older Arm designs (JG-SEC)
  - C vs. RTL only using high-level C perf models
- **Current flow**
  - Combining Theorem-Proving (ACL2) + C vs. RTL Equivalence Checking
  - Exhaustive Proofs for All FP Operations
  - Largely supplanted simulation-based verification of FP Datapath

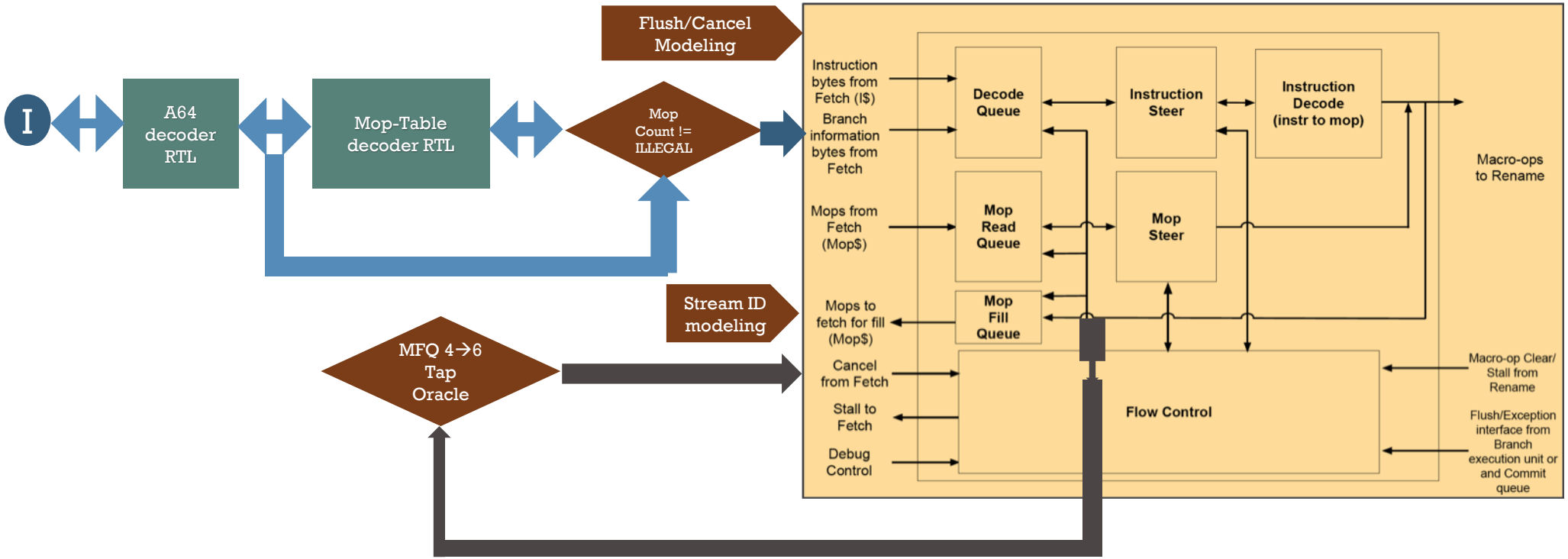
Flow

- Develop custom C model
  - Modularized at lower level of hardware-abstraction
- Prove C model vs. RTL
  - Structural similarity to RTL simplifies C-vs-RTL equivalence check and convergence
- Translate to ACL2
  - Special-purpose parser translates model to ACL2 language (based on *Common Lisp*)
- Encode handwritten proof into a sequence of ACL2 lemmas
- Qualify C model
- Use ACL2's interactive mechanical theorem-prover to check correctness of proof against translated model

II. L0 I-Cache Macro-Op Codec

- L0 I-Cache stores decoded instructions as Macro-Ops
  - Lookup for all fetched instructions
  - Hit saves re-decode cost/delay
- Mop Compression Scheme
  - Decode : Compressed Mop filled into L0 I-Cache
  - Lookup : L0 I-cache entry decompressed into original Mop
- Formal used to exhaustively verify the compress-decompress scheme
  - Formal TB around the I-Decode Unit
  - Found several RTL bugs after all simulation TBs dried up
  - Also found an incidental late RTL bug
    - Unallocated instruction encoding did not generate an UNDEF exception
    - Exposed hole in multiple design and verification methodologies

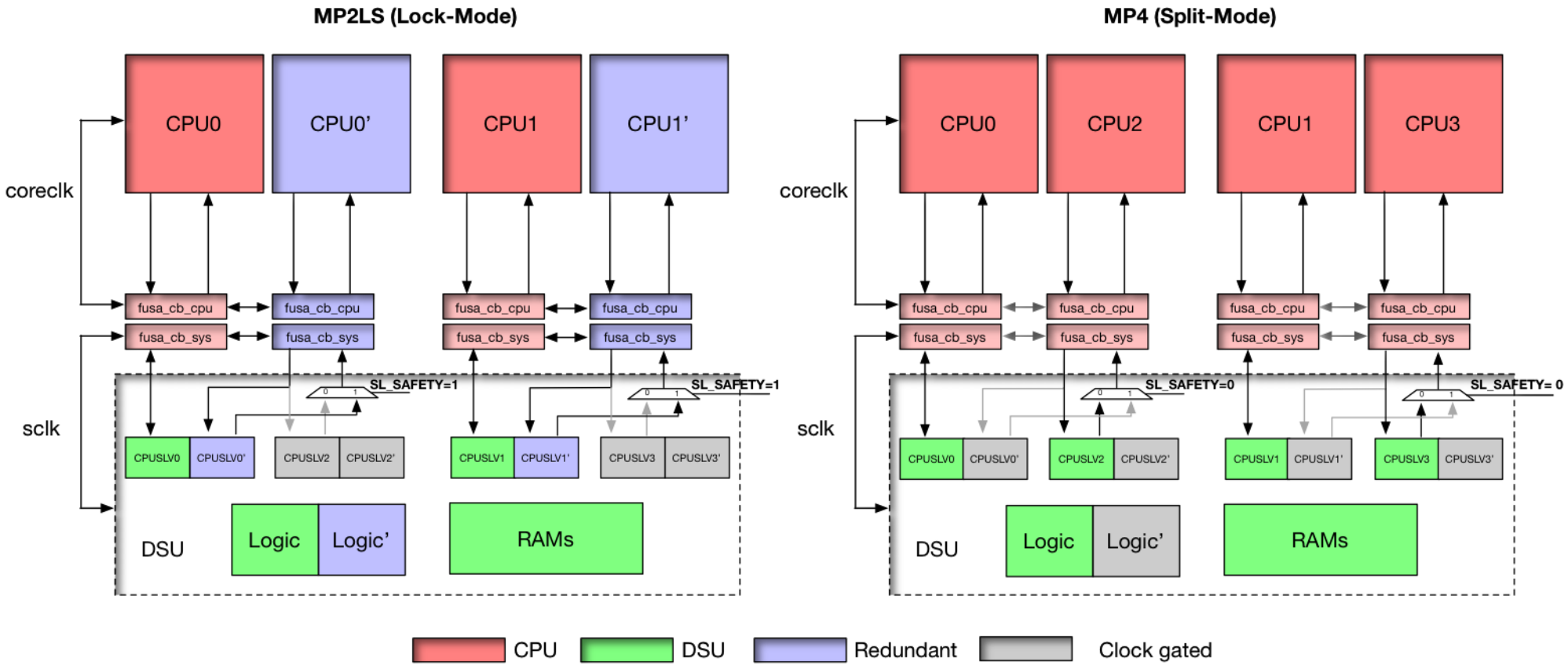
I-Decode Formal TB



III. CPU Lockstep Determinism

- Lock-Mode Determinism in a Split-Lock CPU design
  - 100% PE duplication used for safety redundancy
  - Two CPUs operate in lockstep (separated by a fixed delay)
  - Lockstep errors may occur due to faults or ECC errors
  - Any inherent design non-determinism is undesirable
- Formal used to verify lock-step correctness in absence of faults or ECC errors
  - Sequential Equivalence flow used
    - Manual decomposition
    - Behavioral modeling for RAMs
    - Custom proof-strategies and automated partitioning for convergence
    - Phased proofs – progressively lift simplifying assumptions
- State-space simply too big for simulation to be effective
  - Directed/random testing : bigger compute/resource footprint with limited coverage

Lockstep CPU Design



IV. Vector First-Fault Registers

- **SVE (Scalable Vector Extension)**
  - Predicate-centric Vector Extension to the Armv8-A architecture
  - New Scalable Predicate and Vector Registers
  - Per-Lane Predication
- **Updated Exception Reporting**
  - Faults on select lanes of special predicated SVE ops do not cause architectural aborts
  - A separate *first-fault* register updated instead
  - Sources of faults include both AGU aborts and RAM ECC/poison errors
  - Architectural rules around whether and how this register should be updated
    - Added complexities due to Unaligned and Page-Crossing Loads
  - Originally tested in simulation TB
  - Formal applied late due to designer concerns
    - Found a late corner-case bug

CASE STUDY

System Register Accesses

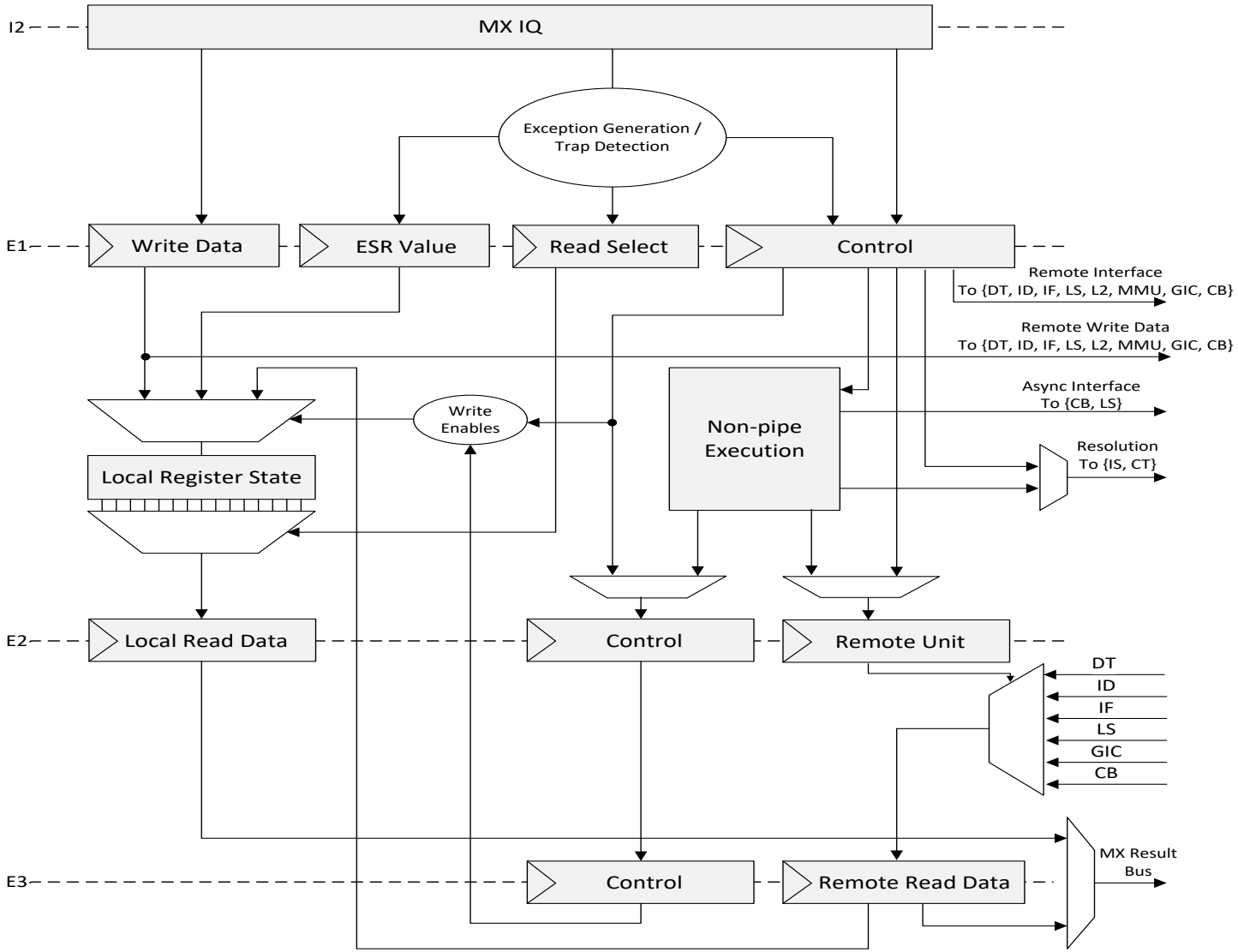
System Instructions

- A subset of "conceptual co-processor" instructions
- Provide system control, configuration and status
- Includes Special-Purpose Registers
  - Current/saved processor-state
  - Link Registers
  - Stack Pointers
  - Floating-Point Status and Control Registers

System Register Unit

- Handles Direct Reads and Writes to almost all architected System Registers
- Orchestrates Indirect Writes to some System Registers
- Enforces Ordering Requirements for System instructions and Accesses
- Detect traps / UNDEF exceptions
- Taxonomy of System Registers
  - **Local**
    - Includes PSTATE registers
    - Updated from a variety of sources
  - **Remote**
    - Located in other units
  - **Sync**
    - Copies across multiple units
    - Pull/Push to/from master/slave copy based on exceptions or synchronization events

System Register Unit

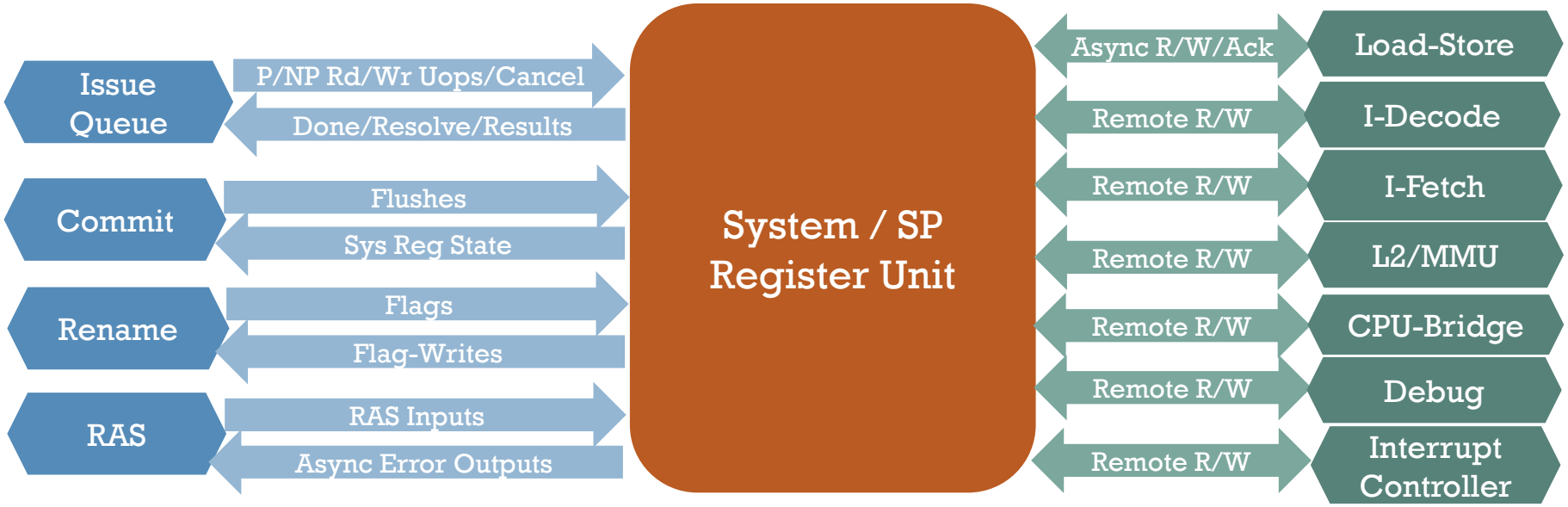


Why Formal?

- A mix of
  - Speculative & Non-Speculative
  - Overlapping Reads and Writes
- Implicit handling around Flushes and Result-Bus Conflicts
- Verification Concerns
  - Correct Ordering (including loss/duplication)
  - Data Integrity
- Functionality exercised in existing higher-level simulation TBs
- Number of corner-case bugs found
  - relatively late, far apart and at a higher level than expected
- Formal expected to converge given DUT size/complexity
  - benefit from exhaustive nature

DUT

Key Interfaces



Ordering Rules

Latency vs. Access	Reads	Async Rd (ARx) inflight	Writes	Async Rd (ARx) inflight
Pipelined (Fixed Latency)	Speculative Request Allowed	• Requests allowed • Response pre-empts Async Rd ACK • Resource Conflict on Response Bus	Request allowed only once Non-Speculative	• Request allowed only if ***ARx*** Flushed • Async ACK for ***ARx*** is Don't-Care
Asynchronous (Variable Latency)	• Speculative Request Allowed • Completion indicated by Async ACK	• Request only allowed when Older ***ARx*** Flushed • Async ACK for ***ARx*** Masked after Request to Remote Unit	Request allowed only once Non-Speculative	• Request allowed only if ***ARx*** Flushed • Async ACK for ***ARx*** Masked after Request to Remote Unit

Interface Modeling

- **Interface Constraints**
  - Qualified in Higher-Level Simulation Testbench
- **Issue Queues**
  - Constrain allowed Ops based on Outstanding Requests
- **Commit**
  - Commit and Flush Modeling
- **Load-Store/CPU-Bridge**
  - Ack Modeling for Outstanding Remote Async requests
  - Fairness Constraints on Responses
- **UID Modeling**
  - ***UID*** : Unique Micro-op Identifier
  - Required Consistency across
    - Issuing and In-flight µ-ops
    - Commit and Flushes

Checkers

- Check **Addresses** of Remote Read and Write requests
- Check **Data Responses** for Pipelined and Async Remote Reads
- Check whether Writes **Correctly Stall**
  - Till they are Non-Speculative
  - Till they can Resolve
- When an Async Access is Outstanding in the Non-pipelined block, the Next younger Async Op **Should Stall**
- Check for **Spurious Read-Done/Write-Done** back to Issue

Checkers (Contd.)

- Any **Remote Read Requests** should be on behalf of **a Valid Micro-Op**
- Liveness Check
  - An Async Read or Write Micro-Op issued **Eventually Results in a Done Response** (and **Data for Reads**)
- **Local Register Data** Checker
- **Remote Write Data** Checker

Complexity Analysis

Formal Complexity

- Total number of flops (DUT + TB): **~6.5K**
- DUT: **4K** flops, **1.5M** gates
- Interface properties: **550** flops, **6.5K** gates
- E2E properties: **900** flops, **1K** gates

Functional Complexity

- Collision Handling between Responses of Pipelined and Async Reads
- Flushes allow Combinations of Multiple Overlapping Async/Pipelined Rd/Wr Requests
- Processing Responses for Flushed Async Reads
- Interacting Stall Mechanisms
  - Writes waiting to be Non-Speculative
  - Outstanding (non-flushed) Async Access

Execution and Results

- 3 months from planning to closure
- Developed following a limited production release of a derivative CPU
- 1 graduate engineer with zero prior experience in formal

- All properties determined
  - 69 assumptions
  - 424 covers (418 covered, 4 unreachable)
  - 267 assertions (249 proven, 18 counterexamples due to known constraint issues)
    - E2E checkers (16),
    - Interface assertions (97)
    - Embedded assertions (154)

- Found a couple of late RTL issues
  - Also used the TB to qualify correct handshaking behavior with neighboring Commit unit

Example Bug

- Failure in Response Data Checker
  - Classified as Erratum
  - Applied to current derivative CPU as well as the older original (released) CPU design
- Bug Description
  - *"The data returned for a remote pipeline register read from Debug-Trace Unit will be corrupted if it is issued to MX-SPR on the same cycle the results are being returned for the second half of an MRRC to a remote nonpipelined register"*
- Erratum description
  - *"Under certain internal timing conditions, an MRC instruction that closely follows an MRRC instruction might produce incorrect data when the MRRC is a read of specific Generic Timer system registers in AArch32 state."*

Conclusions

- No Shortage of Verification Challenges in High-Performance CPUs
- Critical to Know Where to Apply Formal
- When Deciding :
  - Need Careful Comparative Analysis W.R.T. Simulation
    - Design Size (Flops and Gates)
    - Feature Space Complexity & Hotspots
    - Ease of Formulating Exhaustive Coverage and Corner Scenarios in Simulation
    - Ease of Constraining Interfaces in Formal
    - Relative Human/Compute Costs
- Plenty of Scenarios where Success (or lack thereof) with Formal is Predictable
  - No Longer a Wild-Card