RISC-V Integrity:
A Guide for Developers and Integrators

Nicolae Tusinschi, OneSpin Solutions

accellera SYSTEMS INITIATIVE

onespin

2019 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Agenda

- RISC-V Background
- RISC-V Verification
- Operational Assertions
- GapFreeVerification
- Core Verification Results
- SoC Verification Results
- Conclusion

RISC-V BACKGROUND

User-level ISA - Base

- Highly configurable open-source ISA
- 32 bit instructions and 31 fixed-point register with bit width 32, 64, 128 (X0 is constant 0)
- “Base” instruction set “I” (and alternatively reduced version “E”)
  - Usual integer arithmetic/logic, memory, branch/jump, CSR instructions
  - “E” reduces register number to 16 (for smaller embedded systems), only defined for 32 bit

User-level ISA - Extensions

- “M” extension for integer multiplication/division
- “A” extension for atomic read-modify-write memory accesses (AMO)
- “F” extension for single precision floating point (FP)
  - Adds 32 additional FP registers and 3 CSRs
- “D” extension for double precision floating point
  - Needs “F”, wider FP registers
- “Q” extension for quad precision floating point
  - Needs “F” and “D”, wider FP registers
- “C” extension for compressed instructions
  - 16-bit versions of common “I”, “F”, “D” instructions

Privileged ISA - Levels

- 3 potential privilege levels:
  - M(achine)=2'b11, S(upervisor)=2'b01, U(ser)=2'b00
  - M must be implemented and must be privilege level after reset
  - Simple 2 level system can omit S (just implement M and U)
  - S needed for virtual memory
  - Letters "S" and "U" used to capture supported privileges in feature string

Privileged ISA - CSRs

- Privilege and rights of CSRs encoded in upper 4 address bits
  - [11:10]==2'b11 encodes read-only (others read-write)
  - [9:8] encode lowest level where register is accessible
  - Access to non-existing CSR or write to read-only CSR or access to register from higher privilege level causes illegal instruction exception
  - Some CSRs have explicit partial access to lower levels
    - mstatus (0x300), sstatus (0x100), ustatus (0x000)
    - 3 "different" registers implemented in single register

Privileged ISA – misa Register

- Encodes RISC-V string of supported features
  - One bit per letter “A” to “Z” (bits 0 to 25)
  - Two MSBs encode register width (01 – 32, 10 – 64, 11 – 128)
- Address 0x301 ([11:8]=0011)
  - Only accessible in machine mode
  - NOT read-only in spec!
  - Implementations can support switching off some extensions at runtime

Privileged ISA - Exceptions

- 3 memory exceptions per memory access (fetch, load, store/AMO) –
  - Misaligned address, access fault, page fault
- Illegal instruction
  - Non-existing or reserved opcodes and encodings
  - CSR access rights violations
  - Other instructions in unprivileged mode (call/return)
- Breakpoint (fetch, load, store of debugged address)
- Environment call
  - 3 separate exceptions based on originating mode (M,S,U)
- Total of 9+1+1+3=14 different exceptions

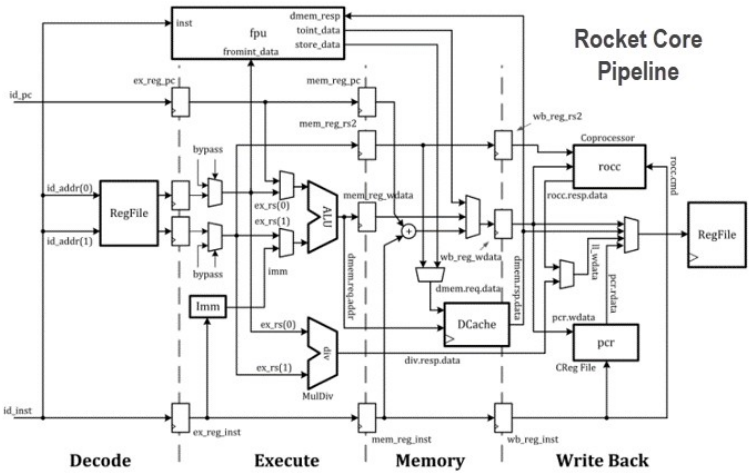
Dimensions of RISC-V Complexity

- Many optional features (covered in previous slides)
- User extensions allowed, including custom instructions
- Designed for many different implementations
  - Pipeline stages, out-of-order execution, etc.
  - Many different microarchitectures
  - Must verify the complete design, not just ISA compliance
- Some applications may have strict security and trust requirements
  - Autonomous vehicles, military/aerospace, nuclear power plants, etc.
- Integrity requires functional correctness, safety, security, and trust

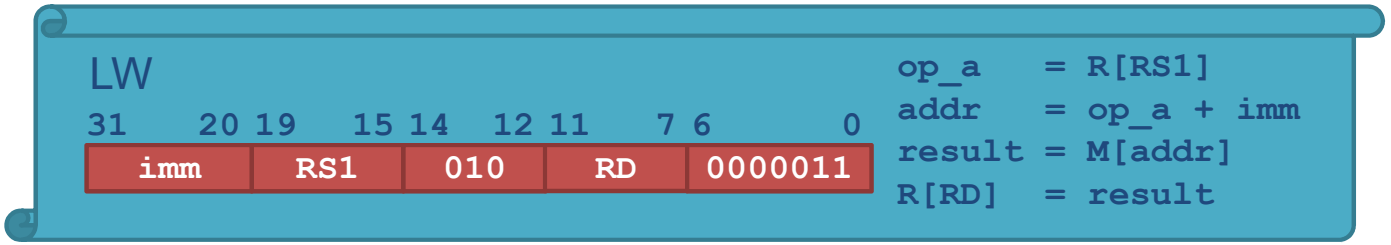
RISC-V VERIFICATION

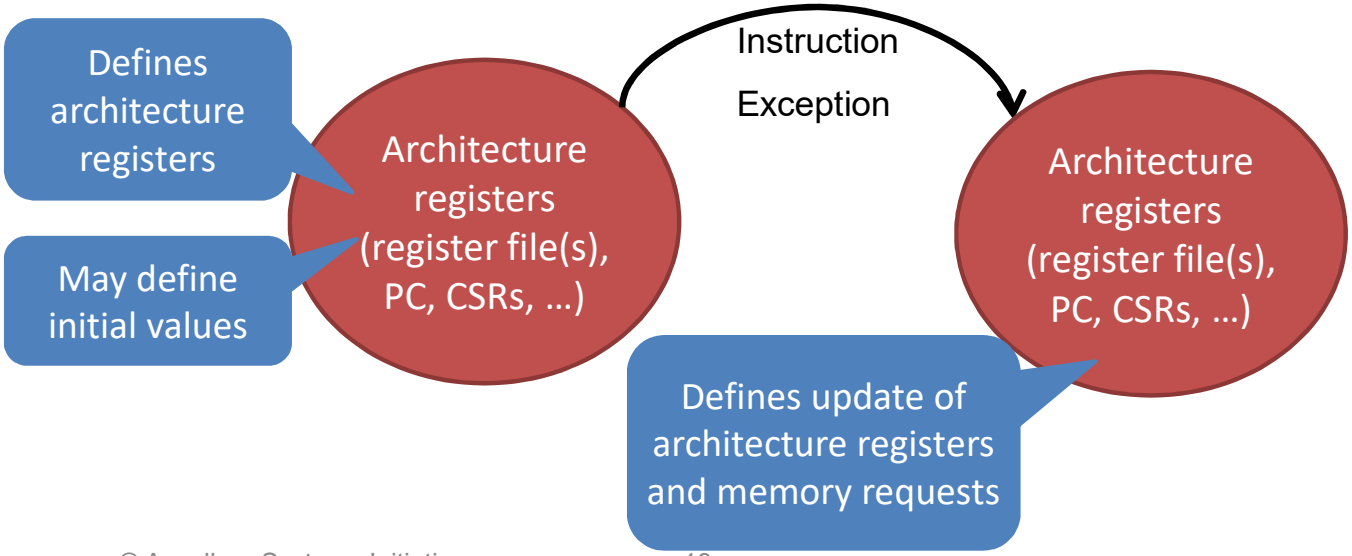
Functional Verification of RISC-V Cores

- RISC-V processor cores are hard to verify
  - Complex microarchitectures to achieve PPA targets
  - Branch prediction, forwarding, out-of-order execution …
- Formal verification
  - Exhaustive verification finds corner-case bugs
  - The only technology that can prove bug absence
- Challenges
  - Complexity issues lead to bounded proofs
  - Hard to write good quality, reusable assertions



RISC-V ISA Specification





Formalized User-Level ISA

- Captures effect of instructions on architecture state and output to data memory
- Formalized in SystemVerilog Assertions(SVA)
- Different extensions such as C, A can be enabled

ISA formalization excerpt for LW

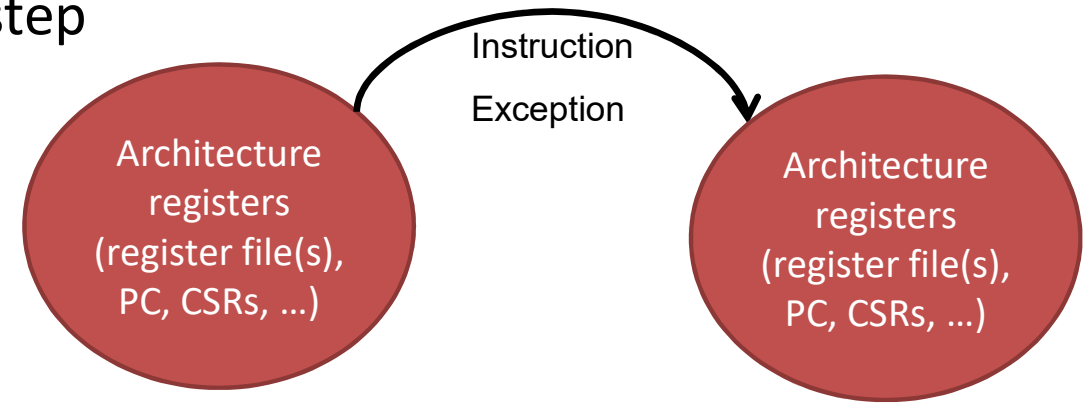
```
32'bXXXXXXXXXXXXXXXXX010XXXXX0000011:
  decode.instr      = LW;
  decode.RS1.valid  = 1'b1;
  decode.RD.valid   = 1'b1;
  decode.imm        = $signed(iw[31:20]);
  decode.mem        = 1'b1;
...
```

Pipelined Microarchitecture

- Various implementation choices
  - Specific pipeline length
  - Forwarding paths to decode state (or additionally also to later stages)
  - Separate ICache/ DCache units with specific protocols
  - Branch prediction for instruction fetch unit
  - Stalling of later pipeline stages or replay mechanism
  - Out-of-order termination for long-latency instructions (like DIV, DCache miss)

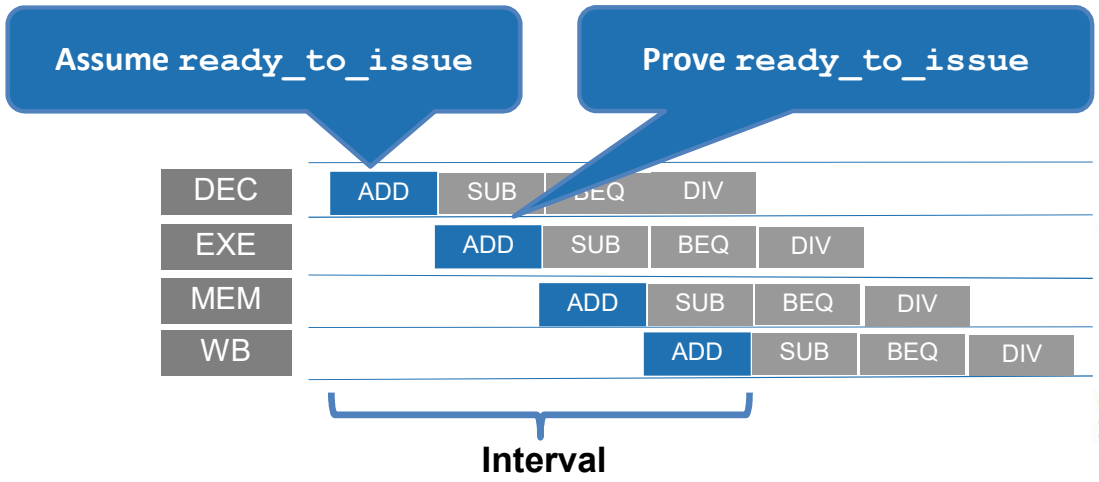
Pipelined Microprocessor Verification

- Link pipeline to sequential execution of instruction
- Capture full effect of one instruction/exception in pipeline one property
- Regardless of preceding or succeeding instructions
- Next sequential instruction “starts” when leaving decode
- Need to capture “sequential” register file where effect of instruction is visible in 1 step



Interval Property Checking (IPC)

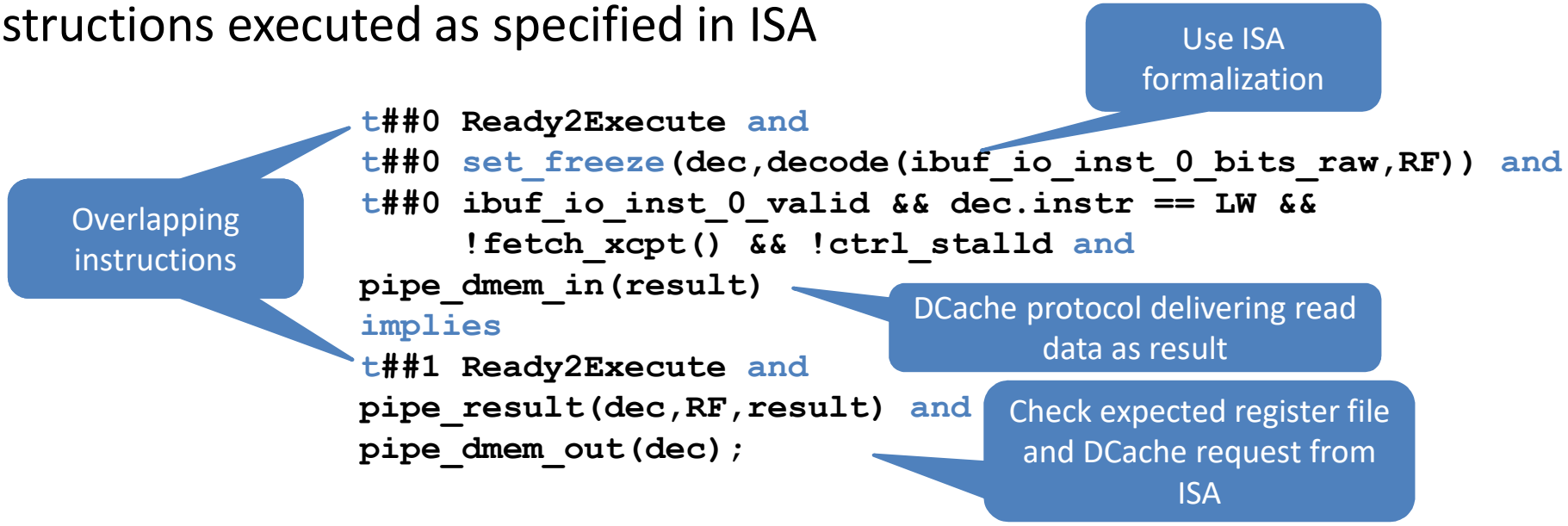
- Reusable SVA achieving unbounded proofs
  - Anatomy of an IPC assertion
    - Does not start from reset but from a generic valid state
    - Limited number of cycles (interval) to reach generic valid state
    - Decouples ISA from microarchitecture




```
property RV32I_ADD;
      decode_t dec;
      logic[63:0] result;
      Arch_state_t cur_Arch;
      t##0 set_freeze(cur_Arch,Arch) and
      t##0 set_freeze(dec,decode) and
      t##0 set_freeze(result, dec.op_a.data + dec.op_b.data) and
      t##0 ready_to_issue and
      t##0 dec.instr == ADD and
      t##0 [...] // no replays, mispredictions, etc.
implies
      t##1 ready_to_issue and
      pipe_result(dec,cur_Arch,result) and
      pipe_no_dmem and // no dmem access
      pipe_no_mispredict(dec) and
      t##1 right_hook;
endproperty
RV32I_Rtype_a: assert property (disable iff (reset) RV32I_Rtype);
```

Verification of RISC-V Implementation

- Instructions executed as specified in ISA



  - Several opcodes can be handled in same property
  - Exceptions, bubbles, and replay handled in separate properties

Verification of RISC-V Implementation

- Instructions executed as specified in ISA
  - Example: Operational SVA for LW instruction fully verifying forwarding to decode/execute and full register update

```
t##0 Ready2Execute and
t##0 set_freeze(dec,decode(ibuf_io_inst_0_bits_raw,RF)) and
t##0 ibuf_io_inst_0_valid && dec.instr == LW &&
     !fetch_xcpt() && !ctrl_stalld and
pipe_dmem_in(result)
implies
t##1 Ready2Execute and
pipe_result(dec,RF,result) and
pipe_dmem_out(dec);
```

Use ISA formalization

Overlapping instructions

DCache protocol delivering read data as result

Check expected register file and DCache request from ISA

OPERATIONAL ASSERTIONS

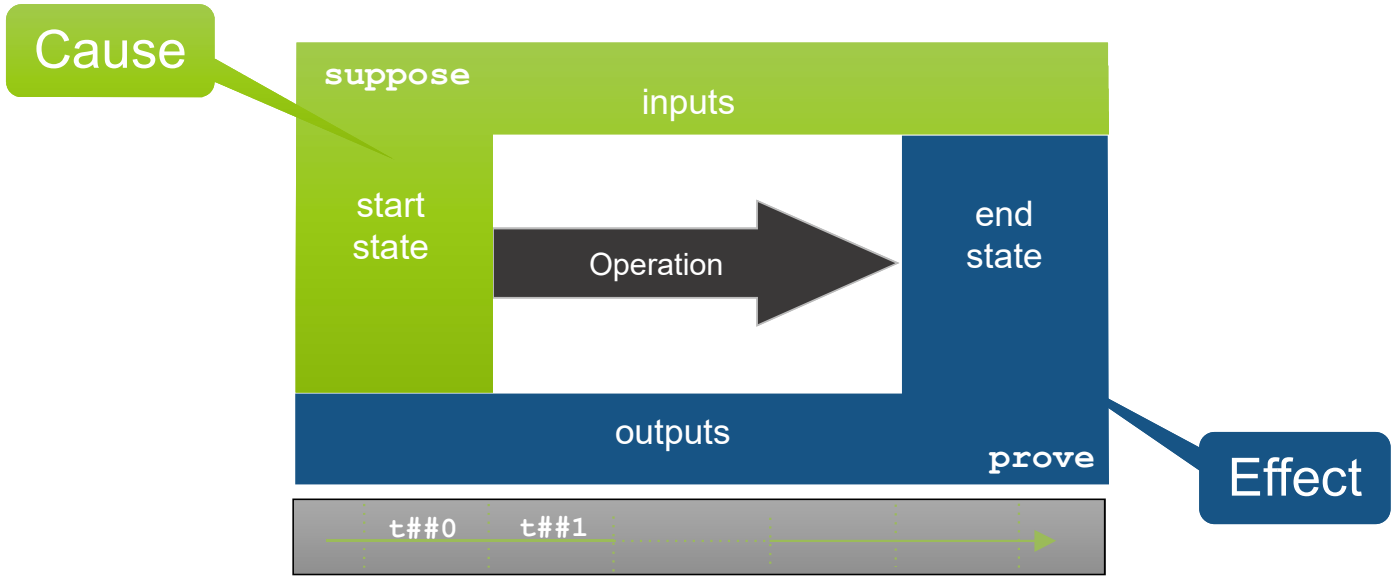
What is an Operation?

- An operation is a multi-cycle activity of the DUV
  - Read or write operation in controller
  - Request is served within n cycles – responsiveness
  - Instruction in processor

- An operation is described by:
  - Start and end state (conceptual, high level of abstraction)
  - Trigger condition(s)
  - Expected output behavior

Operational Assertion

- Formally captures single DUV operation
  - Suppose part describes cause – when does assertion apply
  - Prove part specifies effect - intended behavior in that case



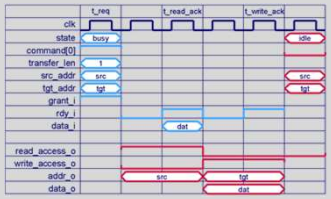
Operational SVA

SystemVerilog Assertions (SVA)

- Expressive, rich assertion language
- Rapid adoption in industry
- IEEE Standard

Timing Diagrams

- Universally used to describe intended behavior of designs
- Familiar to engineers



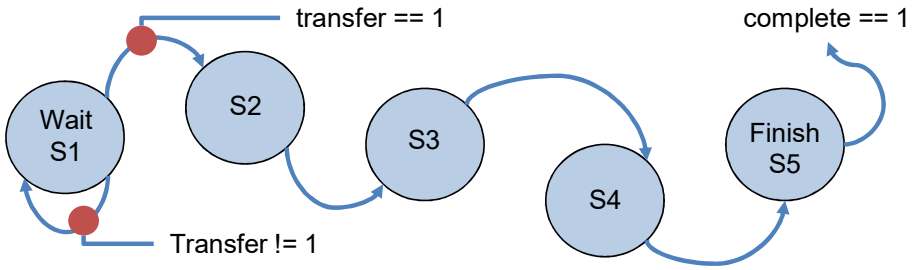
- Describe **cause – effect** relationship
- **Excellent basis for assertion development**

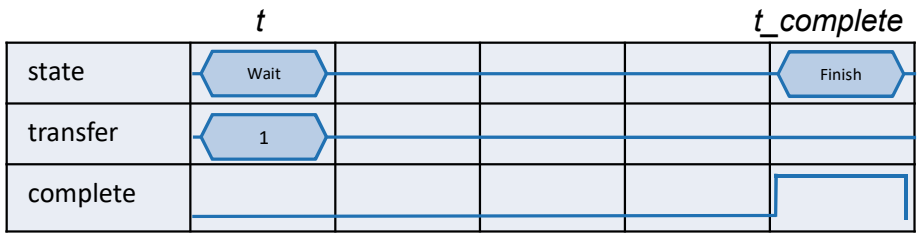
OperationalSVA – SVA Modeling Layer

- Brings timing diagrams to SVA
- Provides predefined SVA macros
- Is standard SVA

Operational Assertions: A Simple Example

Align Operational Assertions with Transactional UVM Sequences






```
sequence t_complete; nxt(t,4); endsequence

property transfer;
   t ##0 state == wait  and
   t ##0 transfer == 2'd1
implies
   t_complete ##0 state == finish and
   t_complete ##0 complete;
endproperty

transfer_a: assert property(transfer);
```

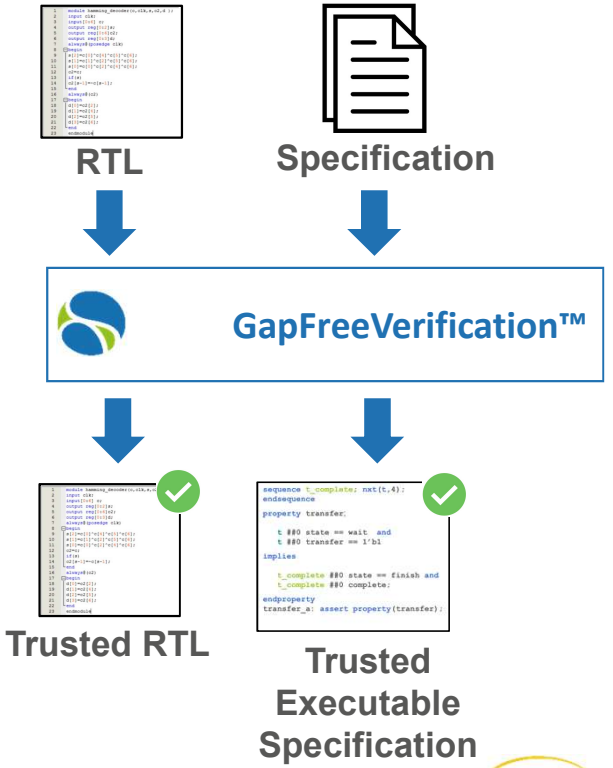
Cause

Effect

GAPFREEVERIFICATION

GapFreeVerification

- GapFreeVerification™
  - Develop executable spec in the form of assertions
  - Prove that executable spec has no gaps or inconsistencies
  - Prove that executable spec and RTL are functionally equivalent
  - Abstraction: specification, RTL
- Benefits
  - Detects errors and inconsistencies in the specification
  - Prove 100% equivalence of spec and implementation
  - Demonstrate absence of bugs/Trojans/ambiguities

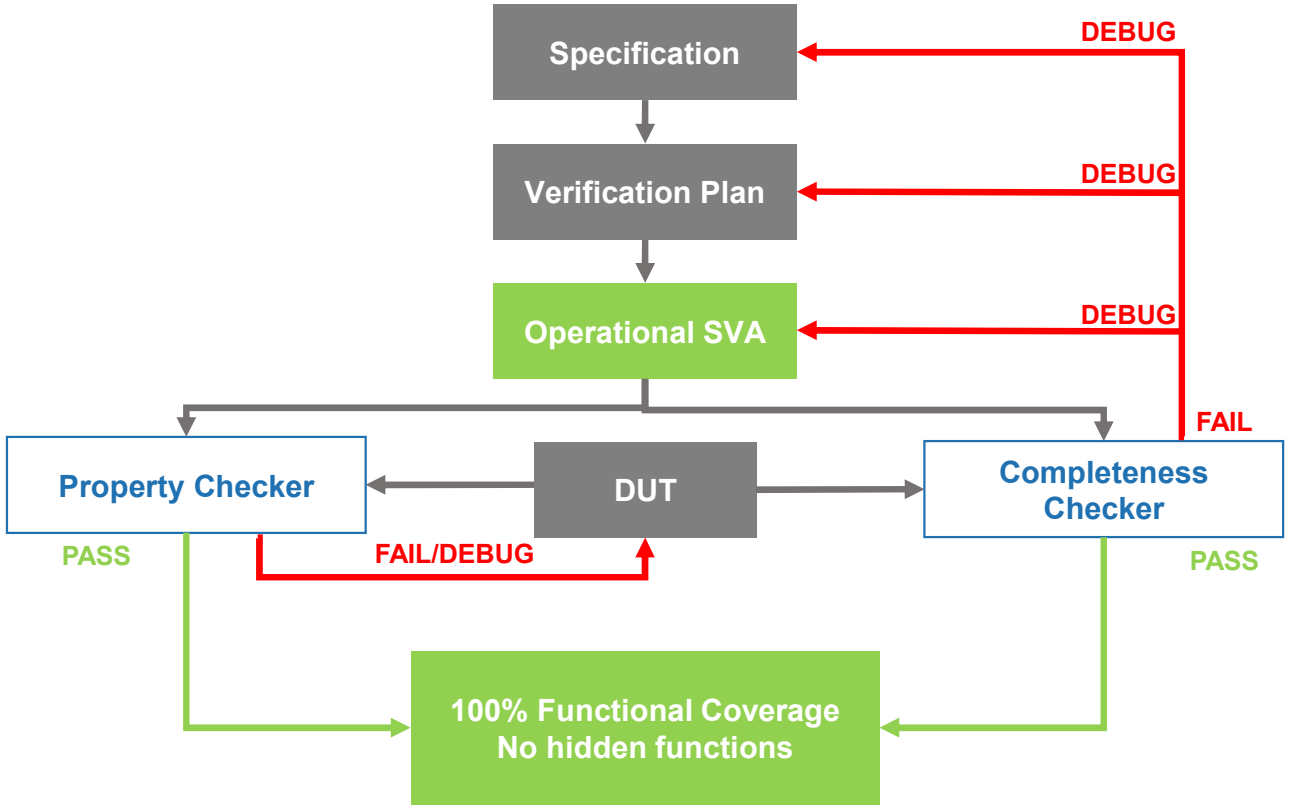


GapFreeVerification

- Achieving 100% functional coverage with SystemVerilog assertions (SVA)

- GapFreeVerification™ rigorous *completeness* definition
  - A set of assertions *P* (formal testbench) is complete if every two designs *C1*, *C2* satisfying the assertions in *P* are sequentially equivalent (for every, arbitrarily long input trace, *C1* and *C2* produce the same output trace)
- Many hardware trust issues are very hard-to-find bugs
  - GapFreeVerification makes no distinction between *"malicious"* and *"naturally occurring"* bugs

GapFreeVerification Process

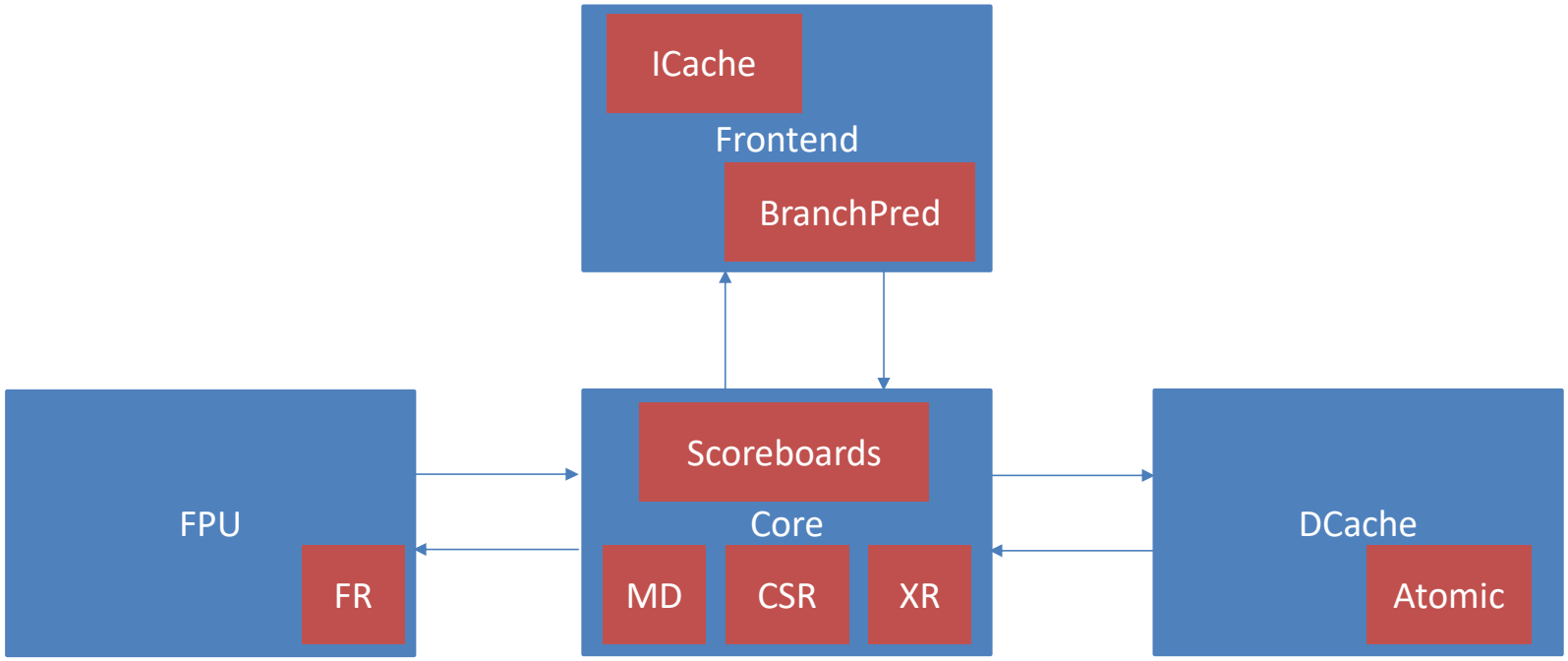


CORE VERIFICATION EXAMPLE

Rocket Core Microarchitecture

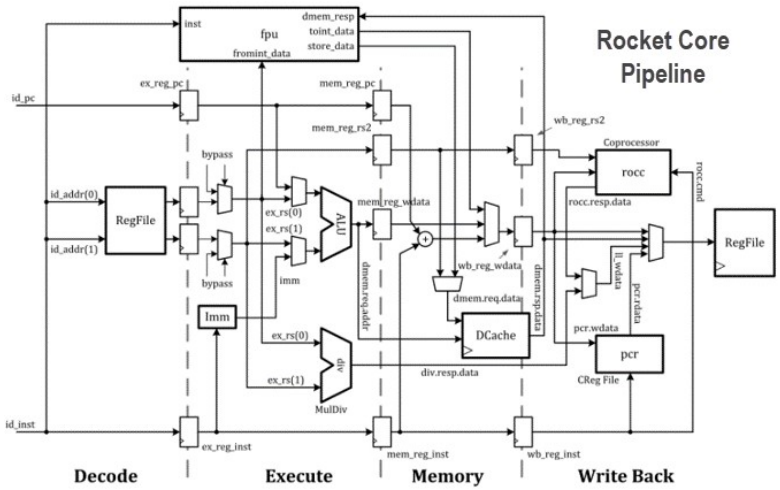
- 64-bit RISC-V core with 39-bit virtual memory address space
  - Includes extensions for integer multiplication/division, atomic read-modify-write, single/double-precision floating-point, and compressed instructions
- 3 privilege levels
- 5-stage in-order pipeline with out-of-order termination for long-latency instructions
- Branch prediction
- No stalling after decode
  - Replay mechanism re-executes instructions on missed handshakes

Rocket “Tile”


ICache
Frontend
BranchPred
FPU
FR
Scoreboards
Core
MD
CSR
XR
DCache
Atomic

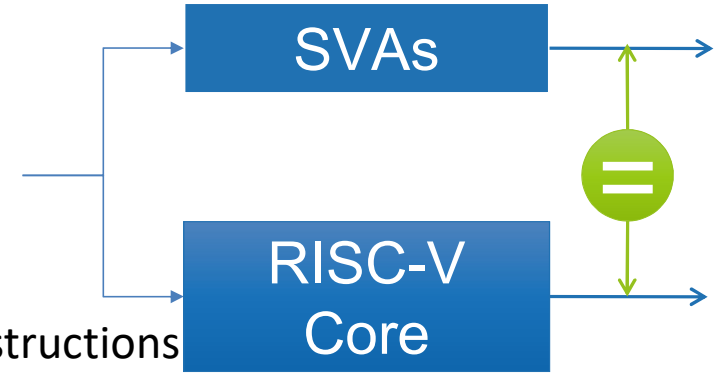
Rocket Core Verification

- 5-stage pipeline, single-issue, in-order pipeline: IF, DEC, EXE, MEM, WB
- Out-of-order completion of long latency instructions (e.g., DIV)
- Branch prediction, instruction replay
- Verified and taped out multiple times



Exhaustive Formal RISC-V Verification

- Leverage SVA formalization of ISA specifications
  - Prove compliance with RISC-V ISA
  - Achieve unbounded proofs
  - Prove bug absence
  - Detect security vulnerabilities
  - Prove absence of malicious logic, including hidden instructions



- Runtime
  - Each property returns a result in less than 10 minutes with helpers
  - Each property returns a result in maximum 5 hours without helpers
- Proof results
  - Each property is reachable and has an unbounded proof result

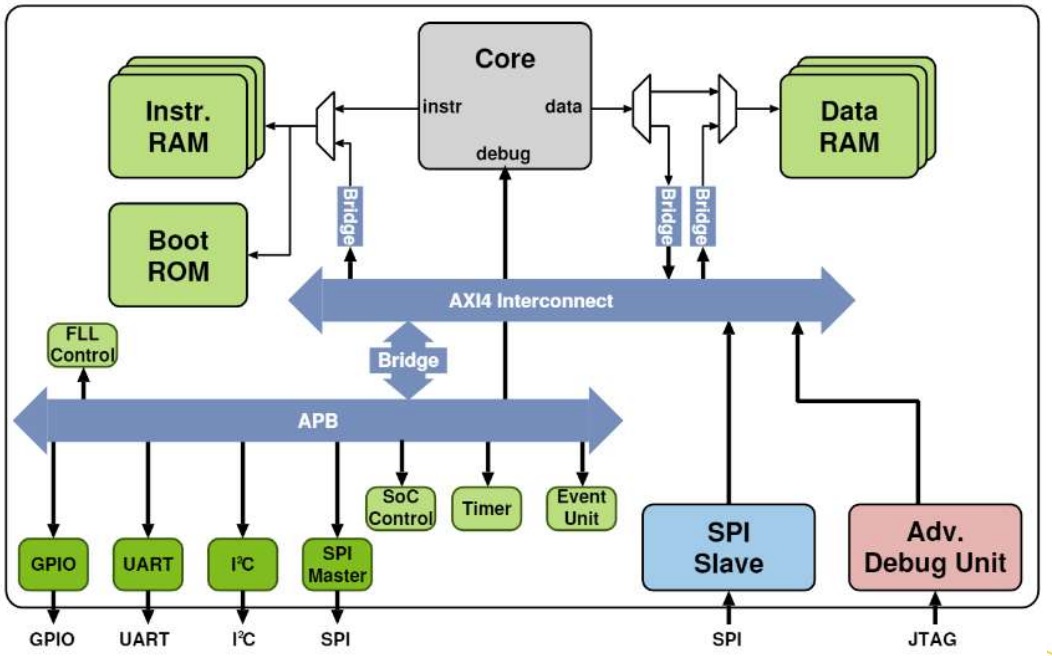
Selection of Issues Found in Rocket Core

- Jump instructions store different return program counter (PC)
  - The instruction fetch unit is responsible to prevent this issue
- DIV (divide) result not written to register file
  - Issue confirmed by Rocket Core developers and fixed in RTL
- Illegal opcodes are replayed (generating memory accesses)
  - Illegal opcodes not generating an exception
  - Issue still under investigation
- Core contains undocumented non-standard instruction
  - Opcode 32'h30500073 (CEASE instruction) not in specification
  - Issue confirmed by Rocket Core developers and fixed in RTL (and spec)
- Return from debug mode is executable outside of debug mode
  - Issue confirmed by Rocket Core developers and fixed in RTL

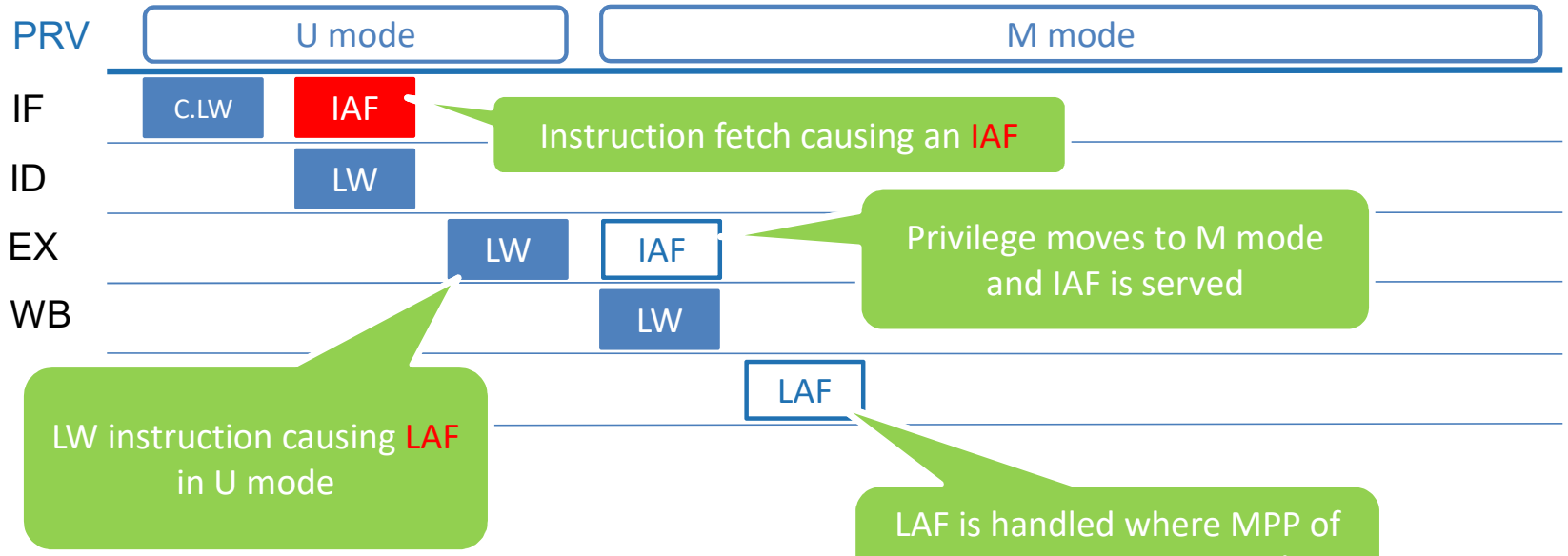
SOC VERIFICATION RESULTS

Parallel Ultra Low Power (PULP) Platform

- Open-source project started by ETH Zürich and University of Bologna
- PULPino Platform
  - Part of the PULP project
  - Single-core SoC platform
- Built for two open-source cores
  - RI5CY
    - 32-bit,4-stage pipeline
  - Zero-riscy
    - 32-bit, 2-stage pipeline
- Rich set of peripherals



Example of Issue Found in RI5CY Core



- IAF – Instruction access fault
- LAF – Load access fault

MPP of MSTATUS CSR written wrongly - (Github issue #132)

Selection of Issues Found in PULPino

- Floating-point addition delivers an incorrect result (-0 + -0)
  - Issue confirmed by PULPino developers and fixed in RTL
- PENABLE signal on APB interface violates address phase protocol
  - Issue still under investigation
- Unique case statement violation results in unexpected instruction decode scenario
  - Issue still under investigation
- Note: verification covered entire SoC design
  - AXI4, APB, and I^2C protocol compliance
  - Wide range of automated checks

CONCLUSION

Summary

- RISC-V cores and SoC can be verified exhaustively by formal means
  - GapFreeVerification approach applies to design with clean specification and "operational" structure
    - Limited number of operations
    - Each computing next architecture state and outputs based on current state and inputs
    - Well suited to detect all undocumented instructions, side effects of instructions, and instruction sequences
  - Verification beyond ISA compliance: microarchitecture/implementation, custom extensions, and absence of hardware Trojans
- Approach has been applied to multiple RISC-V designs
  - Numerous bugs confirmed and fixed by original designers

Thank You!

Any Questions?