Towards Bug Free Application Specific Instruction-Set Processors (ASIPs) with Formal Verification

Oguzhan Turk, Sr. Applications Engineer, Synopsys
Johan Van Praet, Director R&D, ASIP tools, Synopsys
Roy S. Hansen, ASIC Specialist, WSAudiology

Agenda

- VC Formal Overview

- ASIP Designer Overview

- Formal ISA verification Methodology for ASIP Designer

- Formal Verification at WSAudiology

- Summary

VC Formal Overview

Oğuzhan Türk, *Sr. Applications Engineer, Synopsys*

Synopsys VC Formal – Leading Formal Innovations

Unified Compile with VCS

Unified Formal Debugger with Verdi

FPV Property Verification	**FTA** Testbench Analyzer	**SEQ** Sequential Equivalence	**DPV** Datapath Validation
CC Connectivity Checking	**FCA** Coverage Analyzer	**FXP** X-Propagation Verification	**FRV** Register Verification
AEP Auto Checks	**FuSa** Functional Safety	**FSV** Security Verification	**FLP** Low Power

Rich Set of Assertion IPs

ML-Enabled Formal Engines and Orchestrations

Industry's Fastest Growing Formal Solution!

Deliver highest performance
Innovative formal engines and ML-based orchestrations find more bugs and achieve more proofs on larger designs

Enable formal signoff
Exhaustive formal analysis catches corner-case bugs and enables formal signoff for control and datapath blocks

Ease Formal adoption
Easy-to-use formal apps, native integration with VCS and Verdi, and Formal Consulting Services reduce formal adoption effort

Synopsys VC Formal: Innovative Formal Verification Solutions

VC Formal Apps Adoption Effort – Formal Expertise Not Always Required

Low	**Medium**	**High**
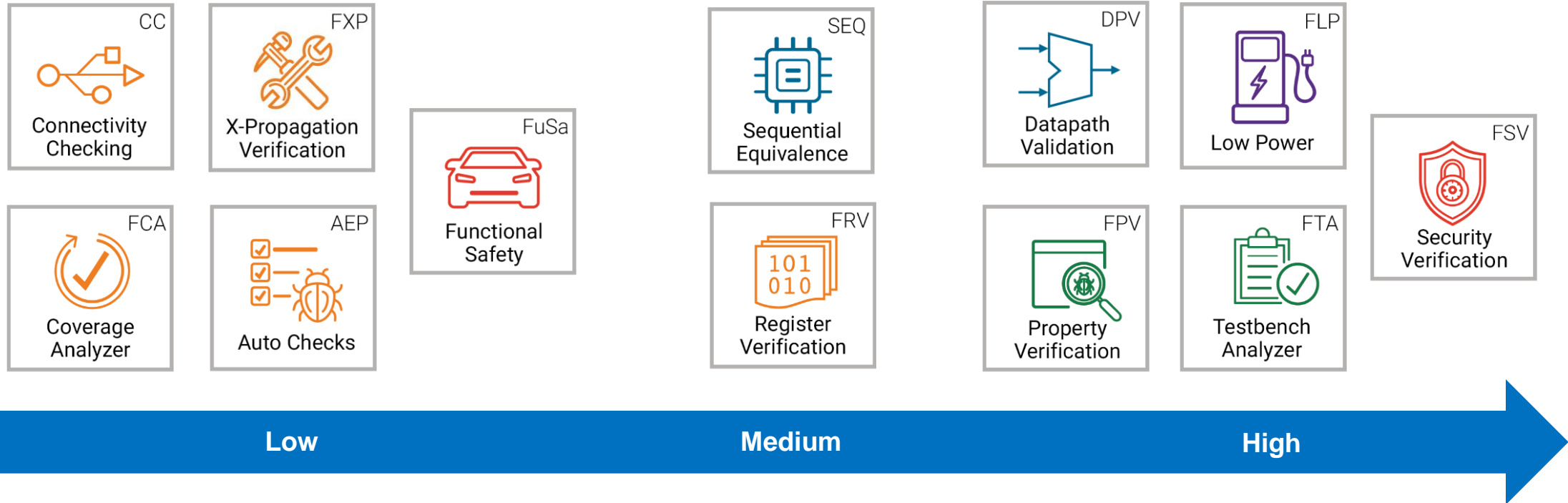
Verification Complexity: In terms of exhaustive computation analysis required to verify the DUT

Adoption Effort: In terms of formal expertise and testbench required to apply the specific APP

Synopsys VC Formal: Innovative Formal Verification Solutions

VC Formal Apps Can Be Used Throughout the SoC Flow

FPV	DPV	FuSa	SEQ	FSV	FLP
Property Verification	Datapath Validation	Functional Safety	Sequential Equivalence	Security Verification	Low Power

FTA	AEP	FXP	FRV	FCA	CC
Testbench Analyzer	Auto Checks	X-Propagation Verification	Register Verification	Coverage Analyzer	Connectivity Checking
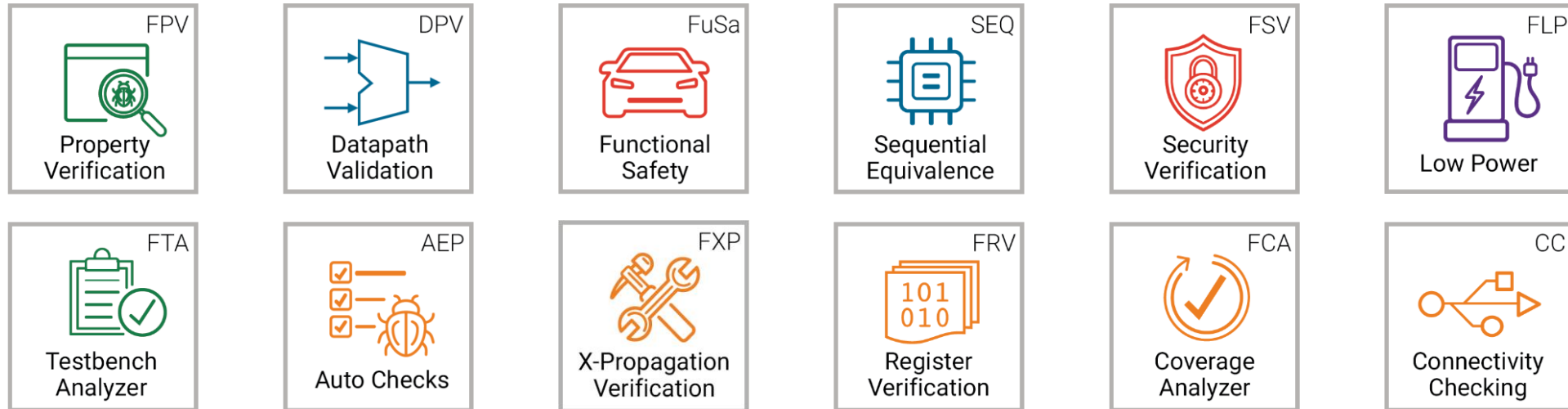
Block/IP ⟶ **Subsystem** ⟶ **SoC**

High Performance: ML powered proprietary engines for hard proofs, liveness, and deep bug-hunting

High Confidence Formal Signoff: Native Certitude integration for fast and high-quality Formal Signoff

VC Formal Leverages Industry Leading Verification Eco-system

Unified Compilation
Common Coverage DB
Integrated UNR

VCS

Verdi

Common Setup and Debug Platform;
Unified Planning;
Integrated Formal Navigator

Common UPF Reader
Unified Low Power Verification

**VC LP
VCS NLP**

**VC
Formal**

VC Z01X

Fault Pruning
Common Fault DB

Lint, CDC, RDC
Formal Analysis

**VC
SpyGlass**

**ASIP
Designer**

ISA Formal Verification
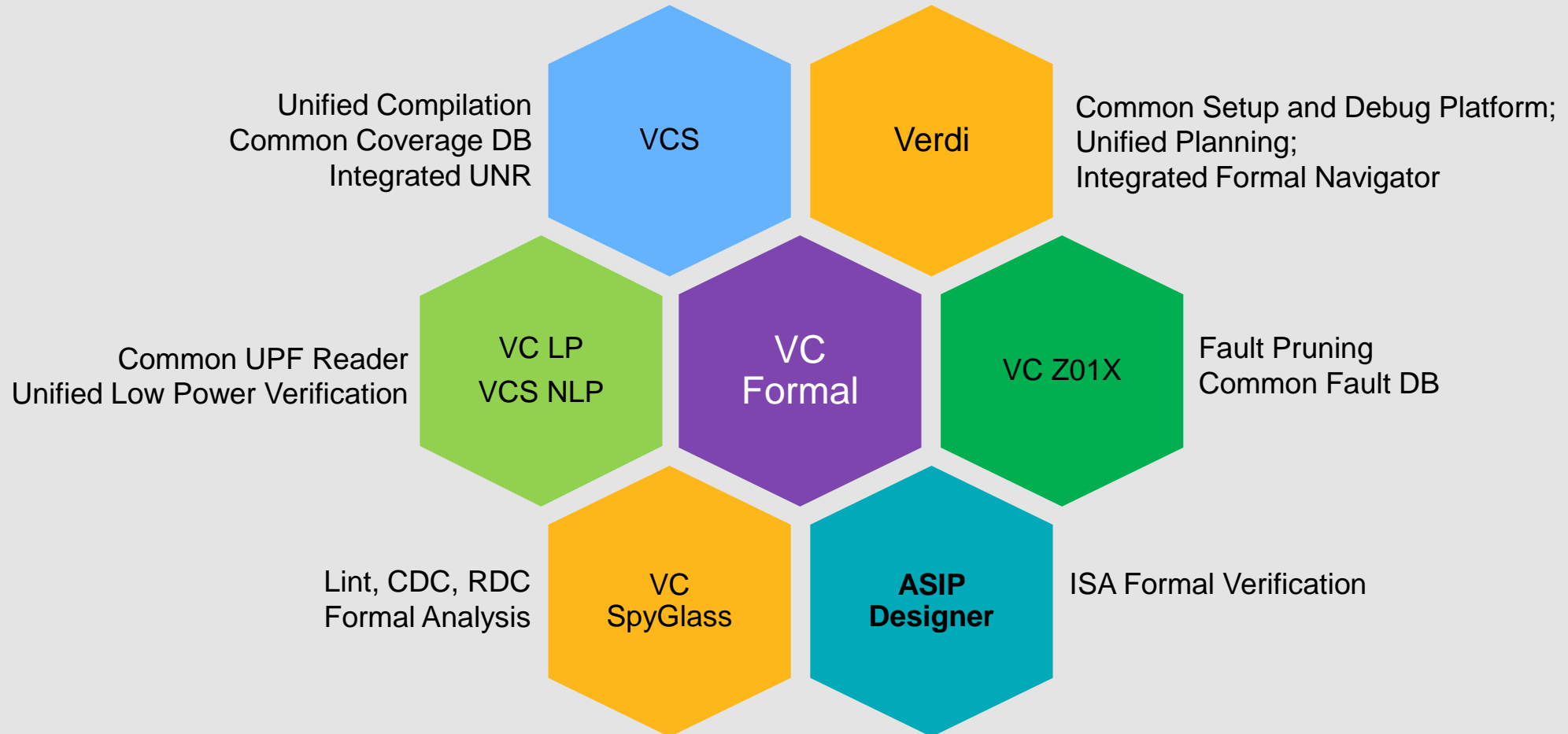
RISC-V Core Formal Verification Overview

- FPV (Model Checking):
  - Prefetch Buffer
  - LSU – Load/Store unit
  - Pipeline
  - RISC-V AIP

- DPV (Equivalence Checking):
  - ALU/MULT/Dotp
  - Decoder

- SEQ (Equivalence Checking):
  - Clock gating verification in every functional unit
  - Designs comparison in presence of new features/timing changes

- FRV (Formal Register Verification)
  - Control and Status Registers (Zicsr)

- FSV (Formal Security Verification)
  - Secure/Non-secure data propagation

Source: https://www.semanticscholar.org/paper/Near-Threshold-RISC-V-Core-With-DSP-Extensions-for-GautschiSchiavone/47f8ce7e0f0f64d0707a13c83c32c30959aa64d5/figure/6
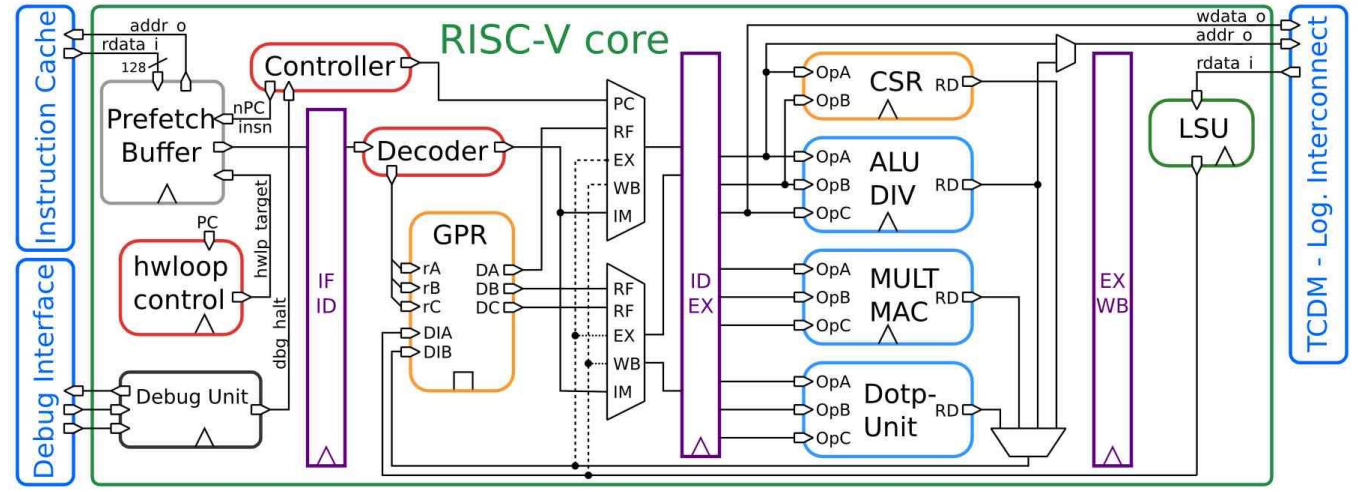
- RV32I base ISA, for example:
  - LOAD - LSU
  - STORE - LSU
  - BRANCH/JUMP/LUI/AUIPC - PFU
  - OP-IMM - EXU
  - OP - EXU
  - Environment call/break point

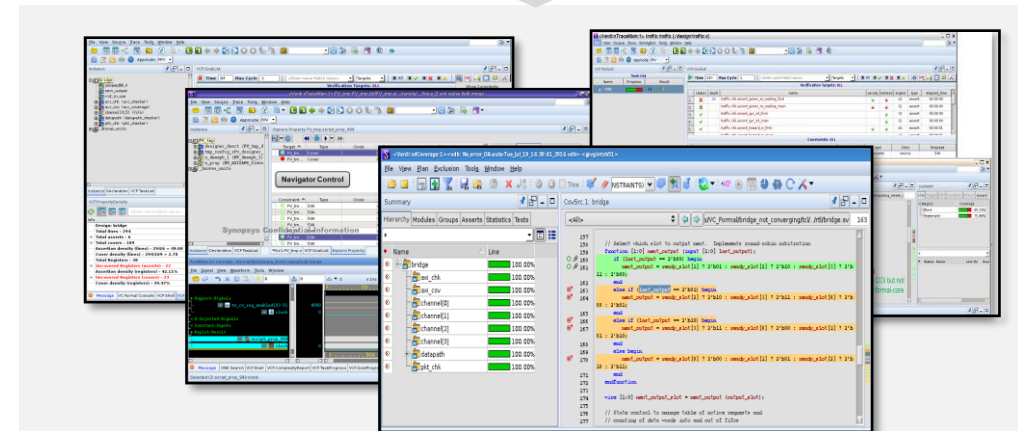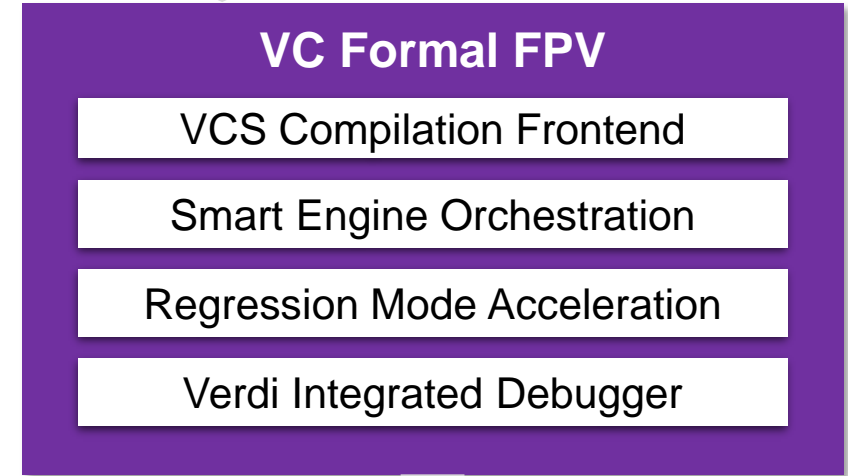- Zicsr extension
  - CSR Write
  - CSR Read

VC Formal FPV: Formal Property Verification

FPV Property Verification

FPV BENEFITS

- Verify functional correctness of design blocks through exhaustive formal analysis

- Find corner-case bugs early without simulation and reduce time to verification closure

- Enable formal signoff methodology

FPV FEATURES

- State-of-the-art ML-powered formal analysis engines and orchestration offer best performance and capacity

- Integrated Verdi GUI offers the most familiar debugging

- Deep bug hunting and advanced proof techniques Proof Assist, Proof Architect
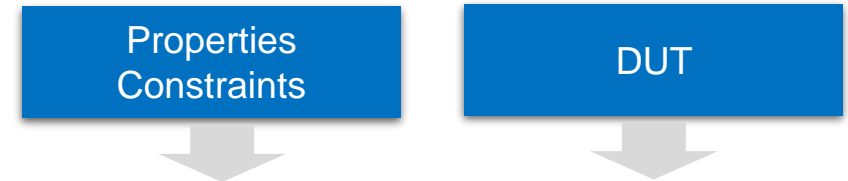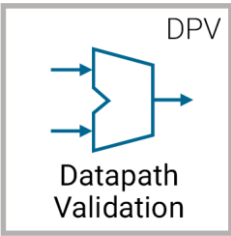
Properties Constraints

DUT

VC Formal FPV

VCS Compilation Frontend

Smart Engine Orchestration

Regression Mode Acceleration

Verdi Integrated Debugger
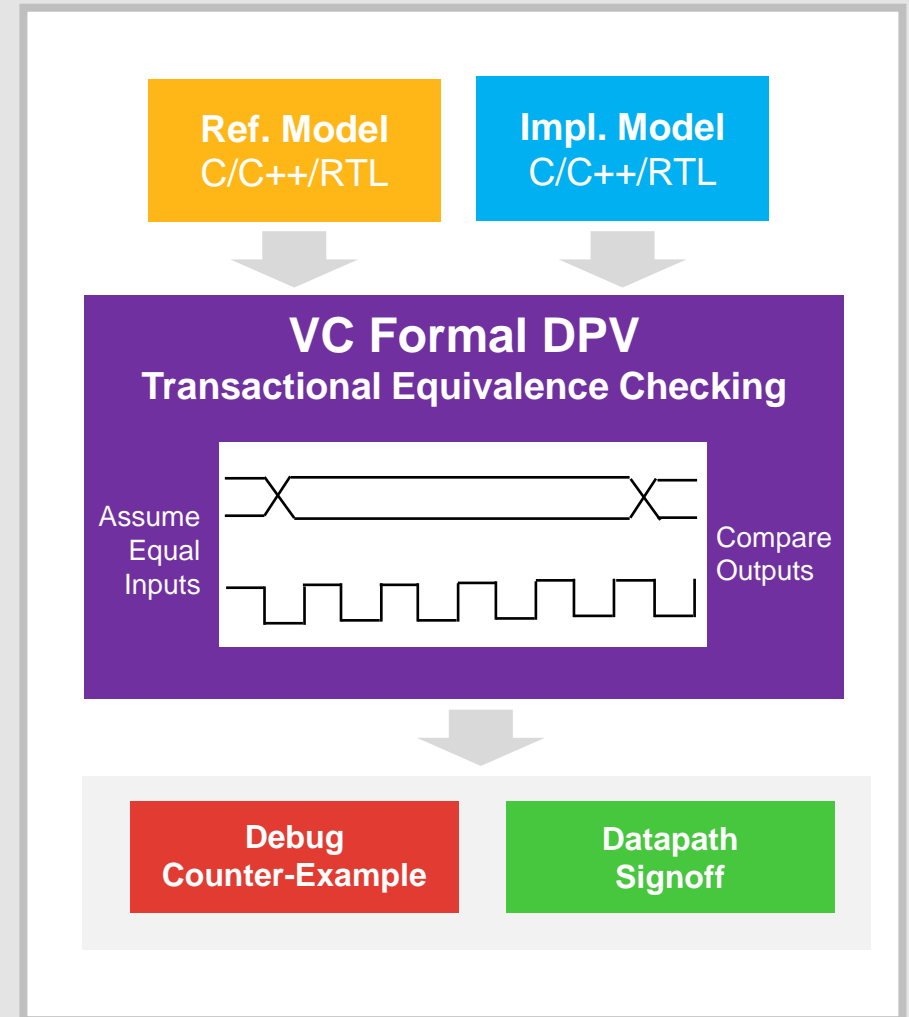
VC Formal SEQ: Sequential Equivalence Checking

SEQ BENEFITS

- Exhaustively verify and signoff the design optimizations without any testbench

- Push the frontier of performance, power, and area (PPA) optimizations

- Save weeks/months simulation regression time

SEQ FEATURES

- Supports clock gating, retiming, microarchitecture optimizations

- Automatically creates equivalence mapping between specification and implementation RTL

- State-of-the-art ML powered formal engine for best performance

Specification RTL → Implementation RTL

VC Formal SEQ
Check Outputs Cycle-Level Equivalence

Debug Counter-Example   **Results Reporting**
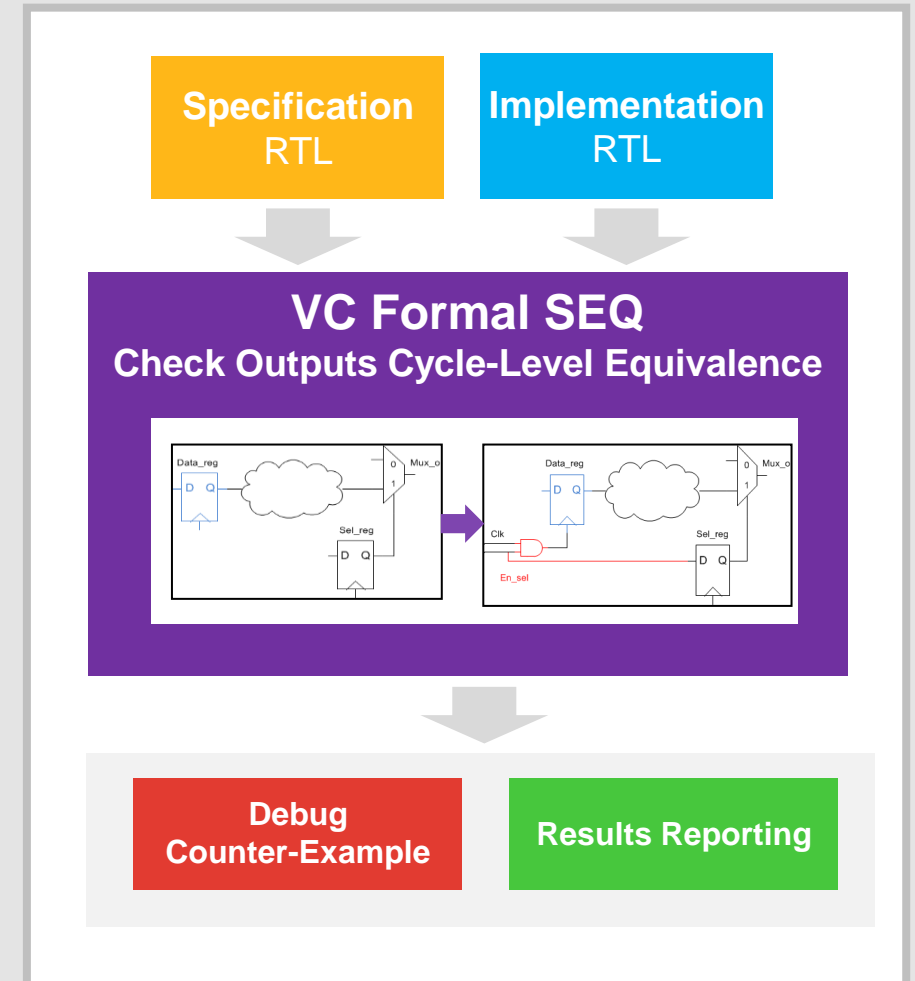
VC Formal FRV: Formal Register Verification

FRV BENEFITS

- Exhaustively verify the consistency of register model against specification

- Find corner-case bugs earlier in the design cycle, shorten debug time

- Save time and effort compared with manual directed simulation tests

FRV FEATURES

- Accept IP-XACT, CSV, RALF spec formats

- Verify that Control Status Registers are correctly implemented using standard or proprietary bus protocols

- Applicable at both the block and SoC level

Register Spec
RALF/CSV/XML

DUT
Register Blocks

AIP
Protocol Constraints

VC Formal FRV

Generate checkers for each register field

Verify register read/write
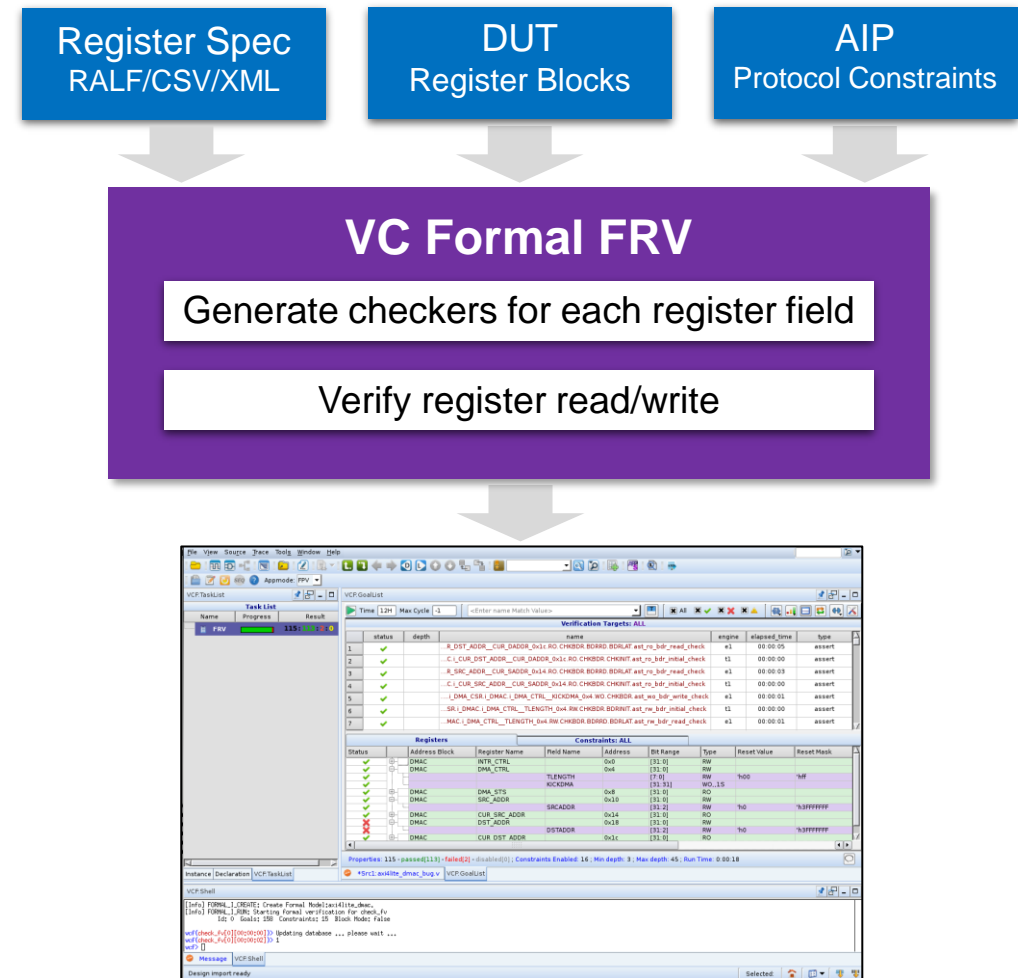
VC Formal FSV: Formal Security Verification
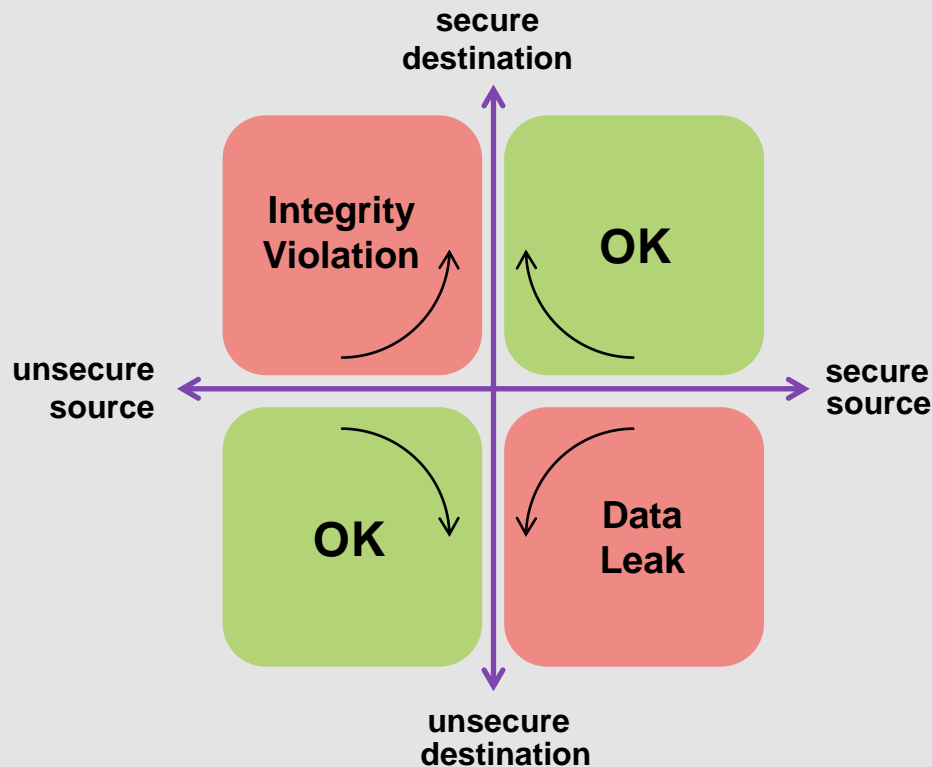
FSV FEATURES

- Flexible property creation & management

- ML powered engines for fast performance

- Data propagation analysis and debug with temporal flow view

- Verification of multiple scenarios in one session

FSV BENEFITS

- Ensure data security objectives are met through exhaustive formal analysis

- Ensure secure data cannot be read illegally or be written from an unsecure source

- Detect security issues that are hard to find through other techniques

VC Formal Differentiations

Performance 7X Faster

Unique Formal Signoff Solution

AI/ML for Performance & Convergence

Industry Standard Datapath Validation

Verification Solution Synergy

Pay-per-Minute Cloud Offerings
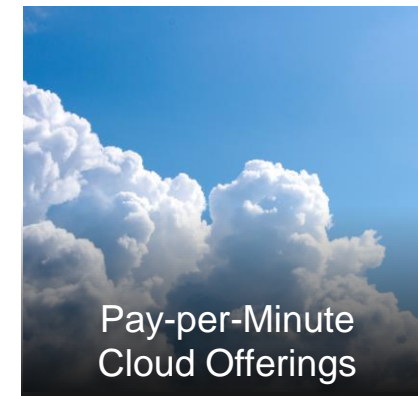
VC Formal Virtual Workshop - Europe & Asia

The Future of Design Innovation: Mastering Formal Verification for Cutting-Edge Designs

Date: Friday, June 21, 2024

Time: 6:30 a.m. - 9:30 a.m. UTC / 12:00 p.m. - 3:00 p.m. IST / 2:30 p.m. - 5:30 p.m. CST

Location: Virtual workshop with hands-on labs. This session is best suited for attendees based in Eastern Europe and Asia.

Registration will close on June 14th. Space is limited!
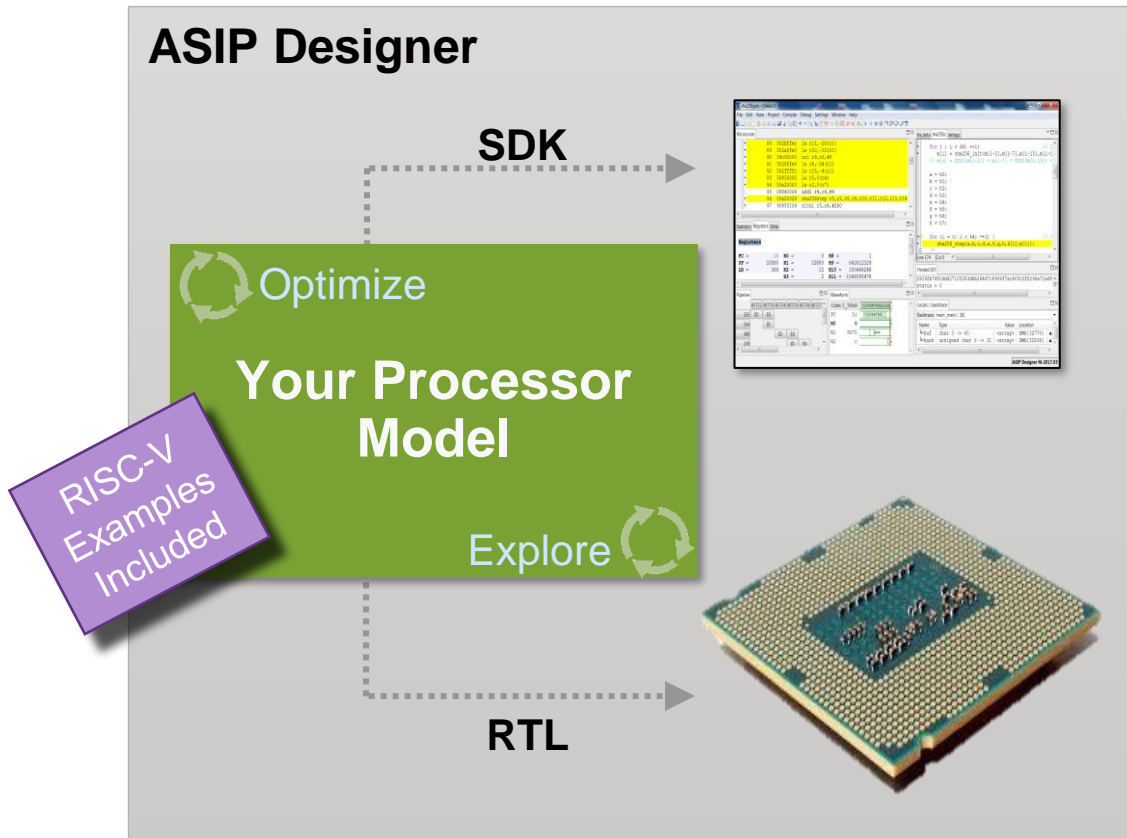
ASIP Designer Overview

Johan Van Praet, *Director R&D, ASIP tools, Synopsys*

Automated Design of Custom Processors

Architectural Exploration with Immediate Tool Support and Immediate RTL Implementation

ASIP Designer

SDK

Optimize

Your Processor Model

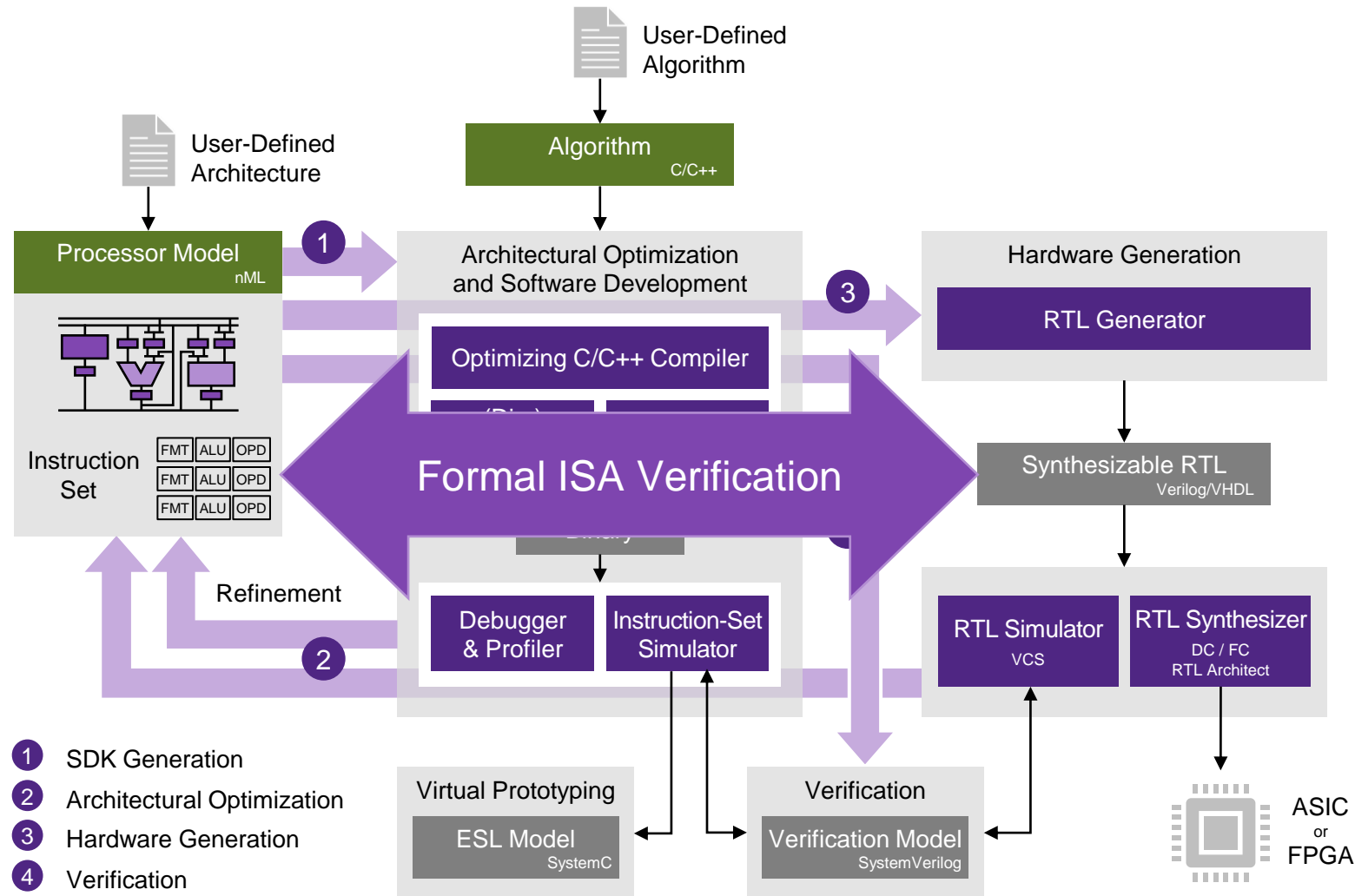**Explore**

RISC-V Examples Included

RTL

- Industry's leading *tool* for creating Application-Specific Instruction-Set Processors (ASIPs)

  - Language-based description of ISA: full architectural flexibility

  - Automatic generation of professional software development kit (SDK)

  - Automatic generation of synthesizable RTL and debug infrastructure

  - Accelerated verification, simulation, and virtual prototyping

  - Integrated with Synopsys' Reference Design & Verification Flows

- More than 2 dozen *example models* included

  - Microprocessors, DSPs, vector processors,…

  - Examples provided in source code, as starting point

Licensed as an EDA tool (not as IP), no royalties
Used by 7 of the Top 10 Semiconductor Developers

See website synopsys.com/asip

ASIP Designer

Tool Flow

Supported design steps

- Modeling of instruction-set architectures: nML language

- Automatic generation of software development kit, including an efficient C/C++ compiler

- Algorithm-driven architectural exploration: **"Compiler-in-the-Loop"**

- Automatic generation of RTL implementation **"Synthesis-in-the-Loop"**

- Design verification
 - Simulation, prototyping
  - Formal ISA verification

Modeling: ISA (nML) + Behavior (PDG)

Instruction-Set & Micro-Architecture   nML

```
// Resource definition
  mem DM[1024]<word,addr>;
  reg RA[2]<word,uint1>;
  pipe C<word>;
  trn A<word>; trn B<word>;
  fu alu;
  ...

// Instruction-set grammar

  opn my_core (arith_inst | ctrl_inst);

  opn arith_inst (a:alu_inst,
    d: div_inst, l:load_store_inst);

  opn alu_inst (op:opcod, x:c1u, y:c1u,
    z:c1u) {

    action {
      stage EX1:
        A = RA[x];
        B = RB[y];
        switch (op) {
        case add: C = add(A, B) @alu;
        case and: C = and(A, B) @alu;
        case or: C = or(A, B) @alu;
        ...
        }
      stage EX2:
        RA[z] = C @alu;
    }
    syntax: op " RA" x ", RB" y ", RA" z;
    image: "0"::op::x::y::z;
  }
  ...
```
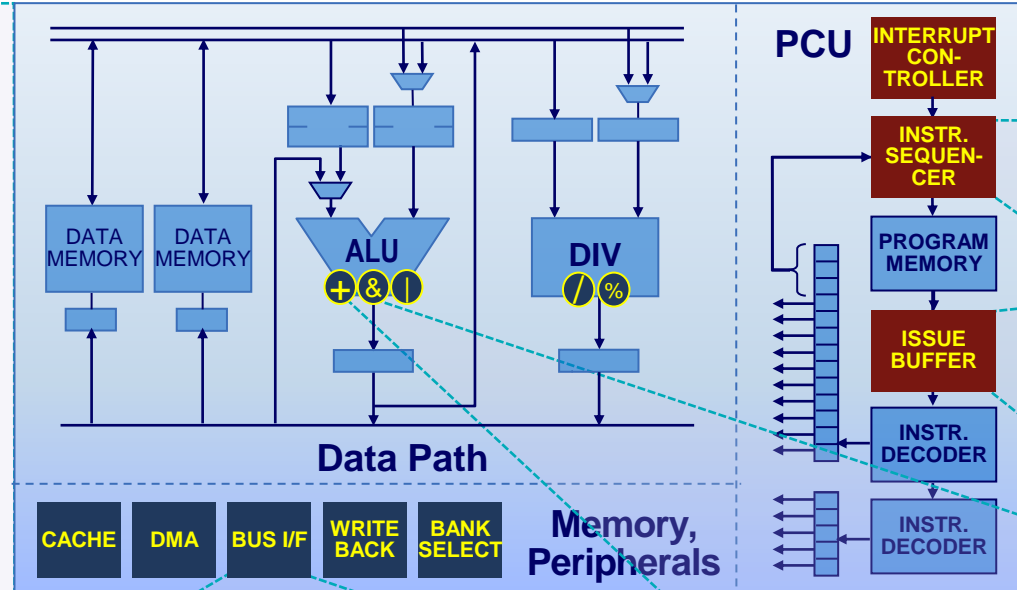
PCU Behavior   PDG

```
void my_asip::
user_next_pc(){
  // manipulation of
  // program counter
}

void my_asip::
user_issue(){
  // creation of issue
  // packets from
  // program words
}
```

I/O Interface Behavior   PDG

```
io_interface my_bus_if(){
  void process_result(){
    // transactions before
    // processor actions
  }
  void process_request(){
    // transactions after
    // processor actions
  }
}
```

Datapath Behavior   PDG

```
// 16-bit saturating addition
word add(word a, word b) {
  int17_t x = (int17_t)a + (int17_t)b;
  if (x > MAX) x = MAX;
  else if (x < MIN) x = MIN;
  return x[15:0];
}
```

Formal ISA Verification Methodology for ASIP Designer
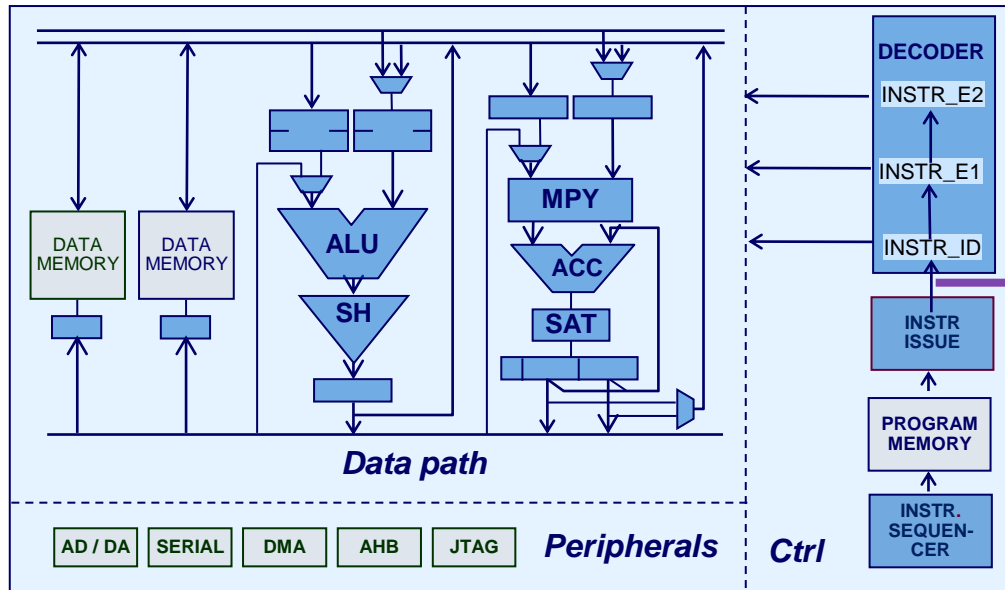
Formal ISA Verification for ASIP Designer (1)

Formally Verify the Generated RTL Implementation Against Expected nML Actions

- Tool generates SystemVerilog properties that express expected behavior at a high, abstract level

  – Requirements are not biased by the implementation

  – Decomposing correctness into manageable claims, for easily converging proofs

- VC Formal FPV proves asserted properties, or finds counterexample

  – A counterexample typically involves parallel activity, corrupting expected action

- ASIP Designer generates assumptions to express invariant tool behavior

  – Assumed properties are generated for the behavior of the C/C++ compiler, to avoid "false negatives"

  – Assumptions generated for FPV are checked as assertions in RTL simulation, to avoid "false positives"

- Verify behavior of data path operations ("primitive functions") separately

  – Use data path verification (VC Formal DPV) for formal verification against an independent reference, if available

  – Tool may replace primitive functions with formal friendly operations for FPV

Formal ISA Verification for ASIP Designer (2)

Generated Properties Use Abstraction

Design

- Peripherals not targeted by generated properties
- Memories are simplified, containing random values
- External inputs are undriven, meaning formal tool can assign arbitrary values

Verification logic applies
single instruction abstraction

Picks arbitrary instruction value
- Tracks it through the instruction pipeline after it enters
- Observes what happens in the architecture, during the lifetime of that instruction

Properties are formulated for tracked instruction

VC Formal FPV proves properties,
or assigns undriven signals and variables
to accomplish the counterexample

Formal ISA Verification for ASIP Designer (3)

Decompose ISA Compliance Requirements for Fast Convergence

- Separate requirements for different phases in the instruction lifetime
  - Requirements for fetch and issue to be provided by the user
  - Generated property for <u>instruction advancing</u> through the instruction pipeline and reaching the write-back stage within N cycles (N depending on pipeline depth, stalls, wait cycles,…)

- Split requirement for <u>result correctness</u>, by instantiating ASIP RTL implementation twice

nML based, unpipelined, functional SystemVerilog translation

Reference equals CPU 1 with instruction in isolation (no pipeline, no parallelism)

Constrained CPU 1
runs isolated
tracked instruction

IF	ID	EX	ME	WB
NOP	NOP	<u>INSTR</u>	NOP	NOP

CPU 1 equals CPU 2 after write-back stage

Unconstrained CPU 2
runs full parallelism

IF	ID	EX	ME	WB
JAL	ADD	<u>INSTR</u>	MUL	SUB

Compute Reference Results

Tool Derives SystemVerilog Reference Code From Each nML Rule With Actions

```
opn alu(d: mRd, s0: mR0, s1: mR1)
{
    action {
    stage EX:
        d = alut = add(alur=s0, alus=s1);
    }
    image : "0000"::d::s0::s1;
}
```

- Simple due to single instruction abstraction
  - Only executes one instruction
  - Purely functional code, no pipelining
  - Execution is aligned with pipeline stages in cpu1, for correct input sampling

```
always @ (*) begin
    if (alu_active && cpu1_sampled)
    begin
        logic [12:0] image;
        if (cpu1_stage == EX)
        begin
            image = cpu1_instr_EX[12:0];
            __R_r_r1_raddr = image[2:0];
            __R_r_r0_raddr = image[5:3];
            __R_r_w0_waddr = image[8:6];
            r_r1 = cpu1_reg_R[__R_r_r1_raddr];
            r_r0 = cpu1_reg_R[__R_r_r0_raddr];
            alus = r_r1;
            alur = r_r0;
            word_add_word_word(alut2, alur, alus);
            r_w0 = alut2;
            cpu1_reg_R_ref[__R_r_w0_waddr] = r_w0;
        end
    end
end
```

tracked instruction is of type alu

it entered the pipeline

input sampling in cpu1

reference result

Generated SystemVerilog Assertions (1)

Instruction Advances and Completes for ALU Rule

```
property cpu2_advances;
  @(posedge clock)
    alu_active && cpu2_seen && !cpu2_sampled
    ##1 !cpu2_kill_ID
    |->
    ##[1:7] (cpu2_sampled && cpu2_stage == WB && cpu2_instr_WB_valid) || instr_killed;
endproperty;
```

tracked instruction seen at input of decoder

- "If the tracked instruction enters the pipeline, it reaches the write-back stage within N cycles"
  - N depends on pipeline depth, stalls, wait-cycles,…
  - A configurable assumption limits stalls, wait-cycles,…

Generated SystemVerilog Assertions (2)

CPU1 and CPU2 Correctness Properties for ALU Rule

- Constrained CPU1 (isolated instruction) compared to reference behavior, derived from nML

```
property alu_correct_cpu1;
  @(posedge clock)
    alu_active && cpu1_sampled && cpu1_stage == WB && cpu1_instr_WB_valid    // Write-Back stage
    |=>
    cpu1_reg_R[__R_r_w0_waddr] == cpu1_reg_R_ref[__R_r_w0_waddr];           // Compare written value
endproperty;
```
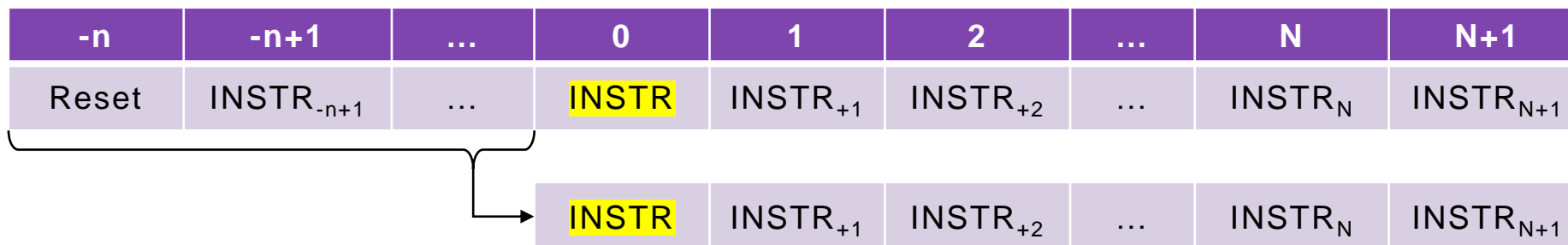
- Unconstrained CPU2 compared to constrained CPU1, aligned to Write-Back stage

```
property alu_correct_cpu2;
  @(posedge clock)
    alu_active &&
    cpu1_sampled && cpu1_stage == WB && cpu1_instr_WB_valid &&           // cpu1 @ Write-Back stage
    cpu2_sampled && cpu2_stage == WB && cpu2_instr_WB_valid &&           // cpu2 @ Write-Back stage
    (cpu2_reg_R[__R_r_r1_raddr] == cpu1_reg_R[__R_r_r1_raddr]) &&        // operand 1 same for cpu1 and cpu2
    (cpu2_reg_R[__R_r_r0_raddr] == cpu1_reg_R[__R_r_r0_raddr])           // operand 0 same for cpu1 and cpu2
    |=>
    cpu2_reg_R[__R_r_w0_waddr] == cpu1_reg_R[__R_r_w0_waddr];            // result same for cpu1 and cpu2
endproperty;
```

Optimizing Proof Times
Use Bounded Model Checking

- Unbounded model checking for "instruction advances" property
  - Acceptable proving times, since the property relies mostly on controller and decoder

- Bounded model checking for "result correctness" properties
  - More complex proving, as this also involves the data path, containing register files etc.
  - Perform bounded model checking with bound N + 2
  - *Without reset,* if result correctness holds under bounded model checking with cycle bound N + 2, it holds without cycle bound too
  - Any longer counterexample can be mapped to a counterexample with length <= N + 2, where in the first cycle the tracked instruction is issued, from an *initial state capturing the history of the long counterexample*

-n	-n+1	…	0	1	2	…	N	N+1
Reset	$INSTR_{-n+1}$	…	$INSTR$	$INSTR_{+1}$	$INSTR_{+2}$	…	$INSTR_N$	$INSTR_{N+1}$

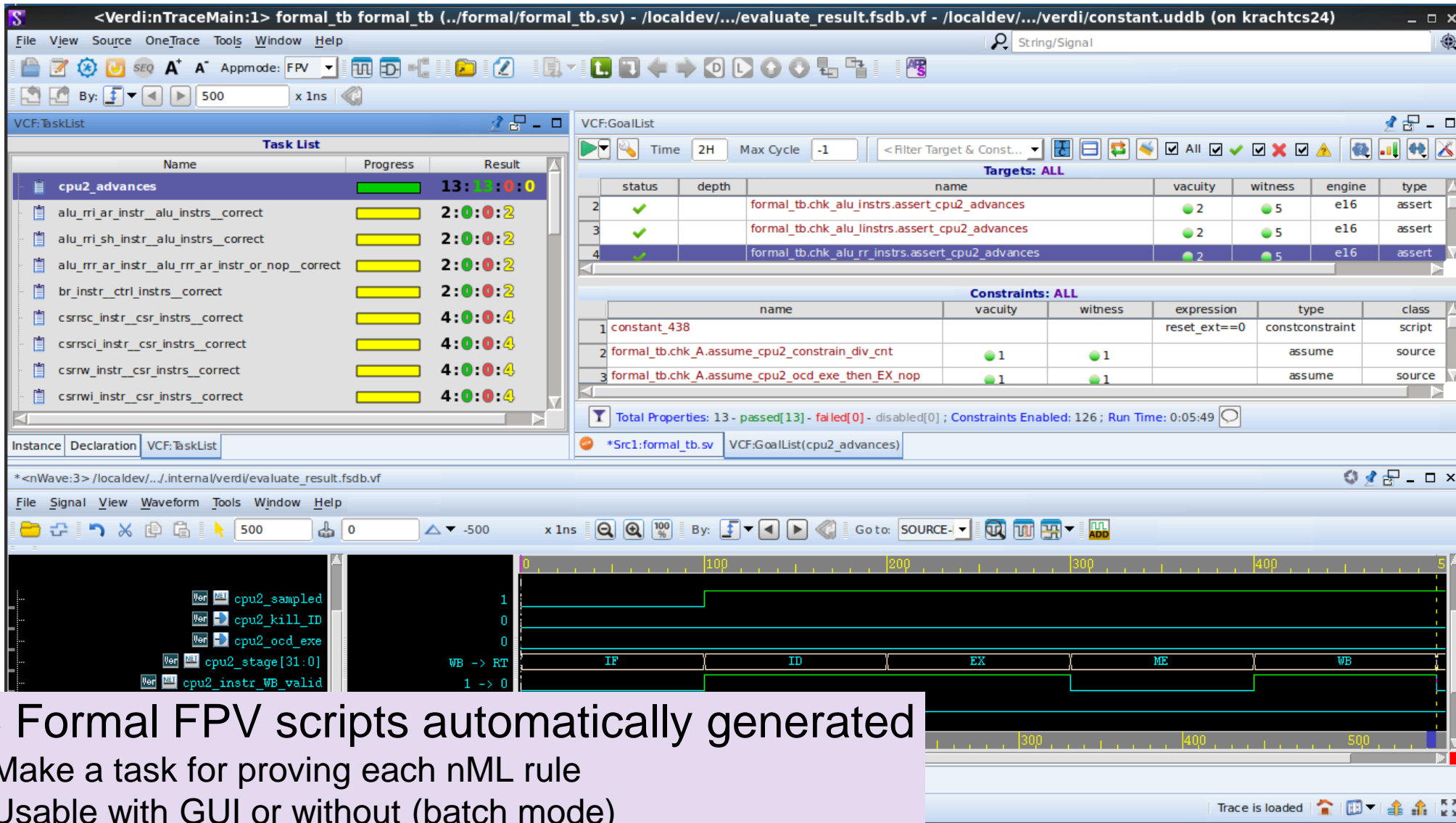|			0	1	2	…	N	N+1
			$INSTR$	$INSTR_{+1}$	$INSTR_{+2}$	…	$INSTR_N$	$INSTR_{N+1}$

Generated SystemVerilog Assumptions

Tool Adds Assumptions to the Formal Testbench to Avoid False Negatives

- ASIP Designer's C/C++ compiler avoids certain instructions or instruction sequences
  - Compiler avoids write conflicts on registers and nets
  - Compiler honors software stall rules
  - Compiler honors control related constraints, e.g. no jumps are scheduled in delay slots of other instructions
- Some general assumptions are needed
  - e.g. on-chip debugging actions only in on-chip debug mode, etc.
  - In absence of reset, some initial decoder states need to be constrained by invariant properties
- User can add extra assumptions to be included in checkers modules

Properties assumed in the formal testbench, are also written out as assertions for simulation

Formal Verification   Simulation

NML → Assertions + Assumptions → Assertions ← Stimuli
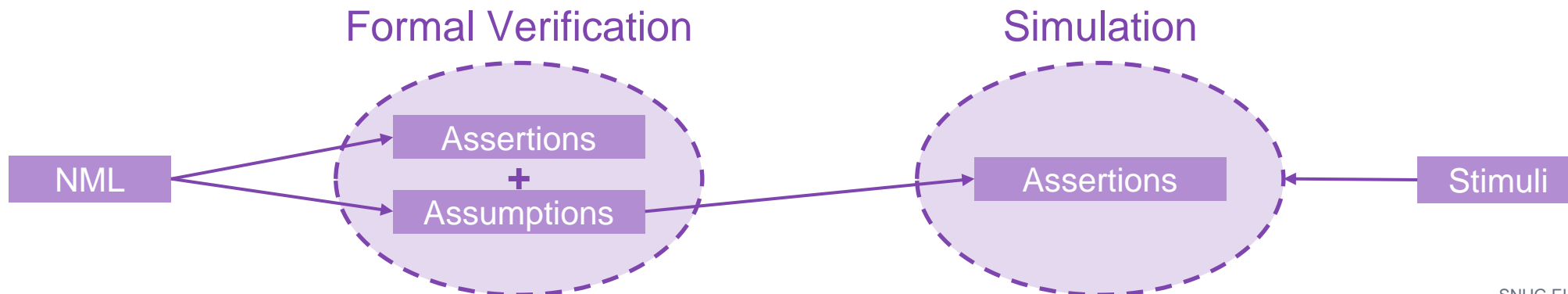
Organization According to nML Rules

VC Formal FPV scripts automatically generated
- Make a task for proving each nML rule
- Usable with GUI or without (batch mode)

Which Bugs Are Found?

- Overall connectivity and data flow in the architecture are verified

- Failing hazard protection
  - On architectures with near-consistent write-back stages per register file
  - Hazards are largely also verified by design tools, but not for e.g. delayed results from multi-cycle units

- Failure to protect instructions from being corrupted by parallel activity
  - Interaction with on-chip debugging and interrupts, delayed results,…
  - This protection is hand-written by the user, and prone to errors

- "Bug Hunting" where FPV complements simulation
  - Some bugs are indirectly connected to generated properties, but *cause corruption of behavior that **is** verified*

- Good results for formal signoff metrics, e.g. with Formal Testbench Analysis (VC Formal FTA)
  - Detection of injected faults in controller (PCU), decoder, hazards logic,…
  - High coverage metric for property density, over-constraint analysis, formal core,…
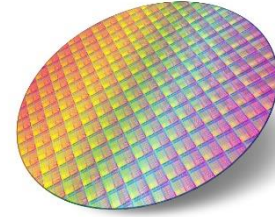
Formal Verification at WSAudiology

Roy S. Hansen, *ASIC Specialist, WSAudiology*

WSA is a global leader in the Hearing Aid industry

2019
Merger of Widex and Sivantos

Widex launched their first digital Hearing Aid devices in 1995 (state machines)

R&D
1000+ engineers develop Hearing Aids and key enabling technologies
R&D hubs in
  Lynge (near Copenhagen), Denmark
  Erlangen, Germany
  Singapore

12,500+
Employees working in more than 45 offices
  HQ Lynge, Denmark

€2,465M
Of revenue FY 2022/23
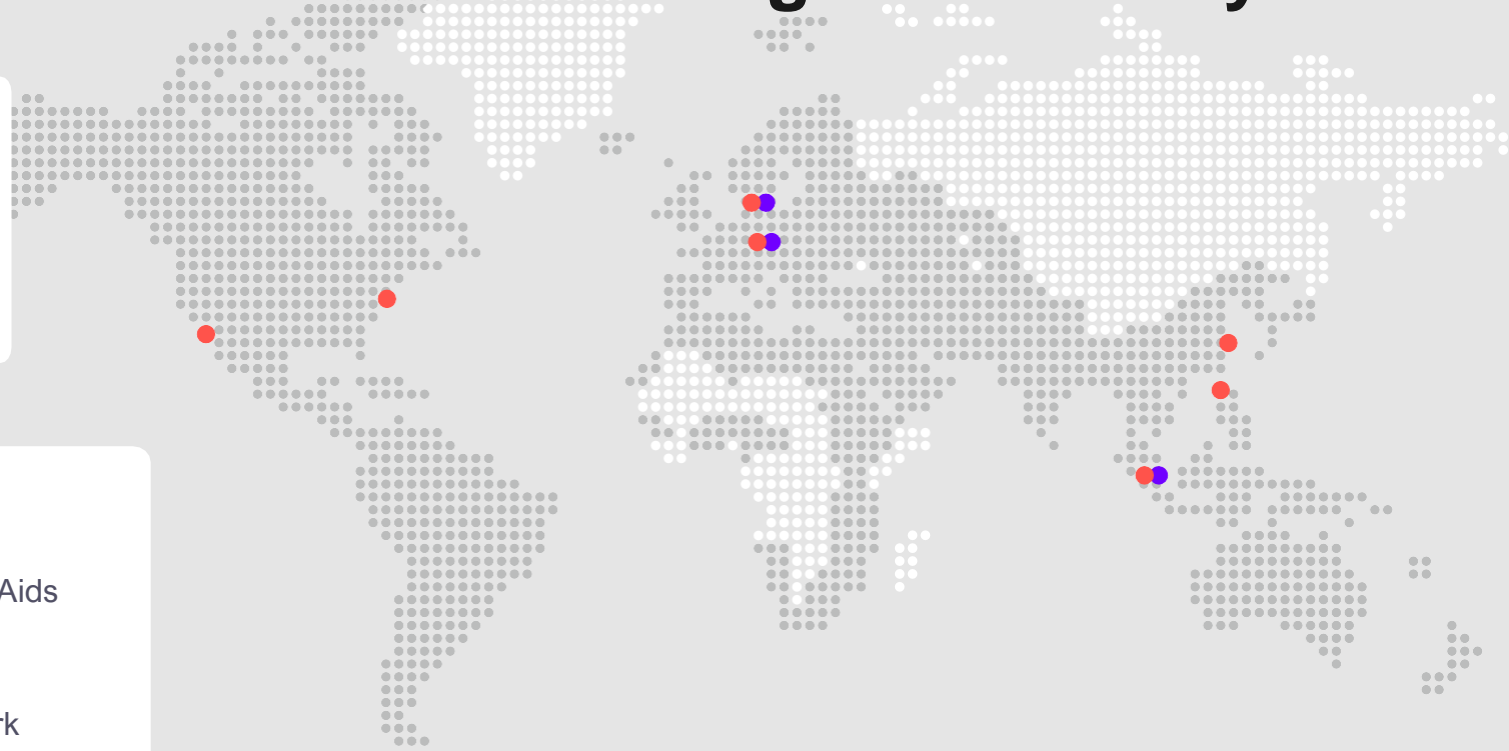Earnings of €480M (EBITA ~14%)
Increased revenue by 50% over the last 5 years
~ 3.8 M people was fitted
Main competitors: GN, Oticon (Demand), Phonak(Sonova), Starkey

IC Development
Team of ~35 Analog and Digital IC designers develop chips that enable our core technology and business

PRESENCE
OUR MAIN PRODUCTION SITES
RESEARCH AND DEVELOPMENT HUB

IC Development for Hearing Aids

- Custom ASIC development to meet power and functional requirements

- Smallest batteries have a capacity of 50 mAh
  - ➢ Must last for 16 hours (= 3 mA)

- Latest process: ~20 nm

- Complexity comparable to first 3G modem chips, but with lower voltage and clock speed

- Rich features:
  - – Bluetooth streaming
  - – Bi-aural digital communication
  - – AI

Analog IC   **Digital IC**
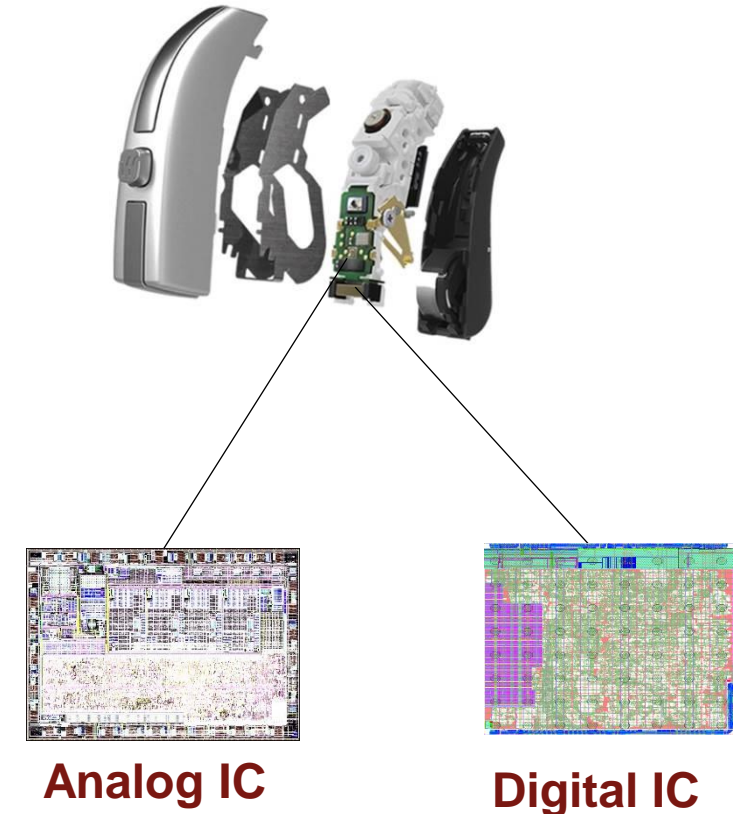
Motivation

"Is there a hard-to-find bug that my testbench has not caught?"

- ASIP Designer Advanced Verification Add-On released in V-2023.12

- I learned about FV techniques back in 1992-95

- I am new to SystemVerilog and SVA

- Time consuming to reach 100% coverage by manually writing testcases

- ASIP Designer built-in instruction-set simulation coverage report does not consider different scenarios (pipeline state, hazards, …)

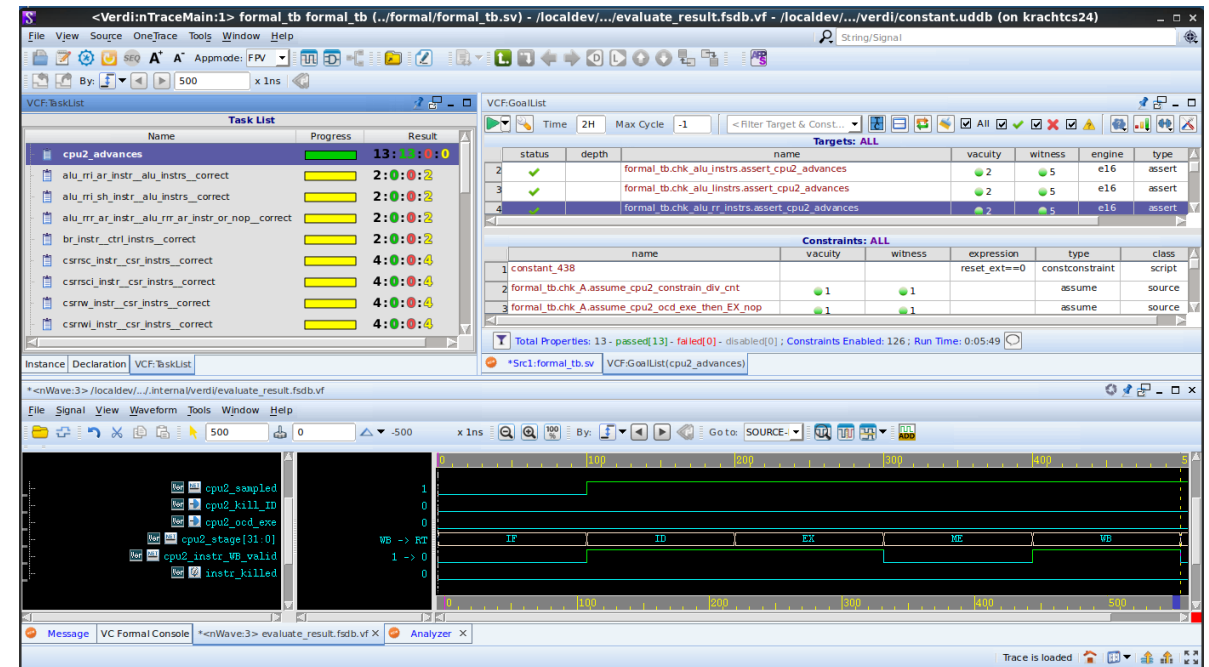- Random instruction generation tool (RISK) does also not provide full coverage

ASIP Features

- Scalar and vector instructions
- Single/Dual/Triple issue of instructions per clock cycle
- Multicycle floating-point division and sqrt units
- System bus inserting processor stalls
- Some vector instructions are vector-guarded

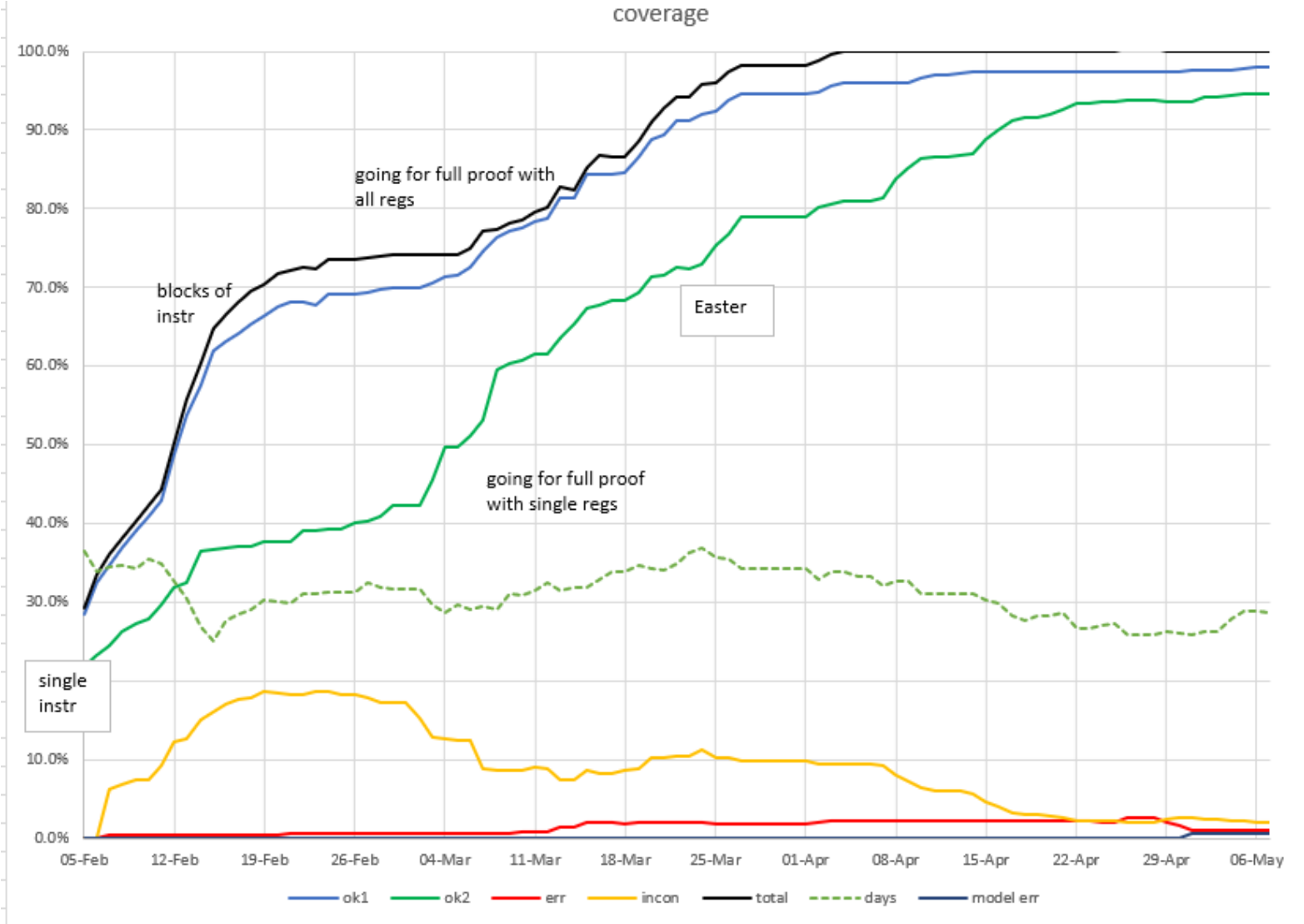➢Last three points add complexity for FV

Getting Started

- ASIP Designer provides a manual and a step-by-step tutorial based on tnano and trv32p5x example cores

- Synopsys assisted to get started with our custom ASIP

- Many tricks used to ensure convergence
  - ➢ Split instructions into multiple tasks and scripts
  - ➢ Use blackboxing of unrelated design parts
  - ➢ Bounded proofs
  - ➢ …

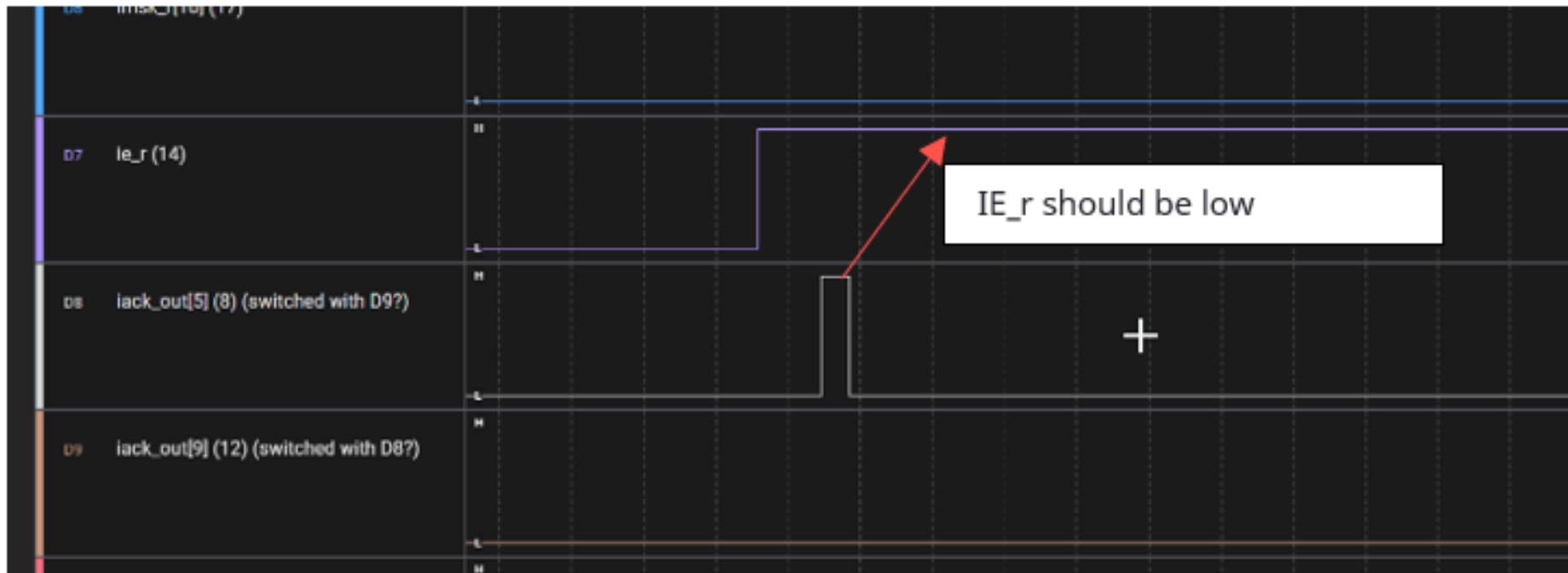- Convenient to run multiple tasks over night

Progress over time

coverage

going for full proof with all regs

blocks of instr

Easter

going for full proof with single regs

single instr

Legend: ok1, ok2, err, incon, total, days, model err

The missing interrupt (1)

- Originally observed on FPGA
- After an acknowledged interrupt, we expect IE to go low, but it did not:

The missing interrupt (2)

- Simple to create and debug a counterexample using formal properties
  - assert property(@(posedge clock) cpu2_irq ##0 cpu2_iack[->1] |=> decode_swi(cpu2_IR_ID));

- Cause: Bug in PCU that clears interrupt when it arrives during a processor stall
- Now detected with auto-generated assertion in newer tool version
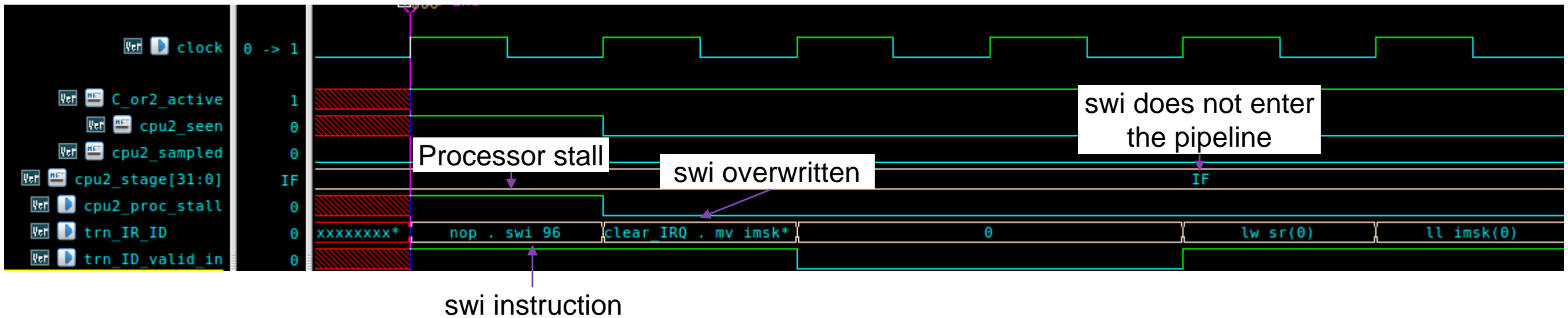
Cache Alignment

- Original: Cache used 20-bit address
- New: 32-bit address type shared with IO interfaces
- Bug: Potential overflow in PCU

➤ In real life, we don't have 2^{20} instructions, so it's not an issue

```
if       (rel_jump)   new_pc = pcr + offs;
else if (abs_jump)  new_pc = trgt;
else if (dlp_jump)  new_pc = lsr;
else                     new_pc = pcr;

new_pc_algn = new_pc[19:1]::0;
                              ↓
                             31
```

Reset of Cache Counter Register CH

- Cache counter is updated inside PCU
- nML rule to reset cache counter
- PCU and nML rule write to CH at the same time
- PCU wins -> reset not executed

```
if (ohe_selector_EX[0]) // (ch_w_copy0_lmr_EX)
    1

begin

  // [moves.n:1143]

  ch_w_out = lmr_in_opdis;
  2            0

end

if (CH_ch_w_loop_buf_process_pdg_en_in)
    1

begin

  ch_w_out = ch_w1_in;
  2           2

end
```
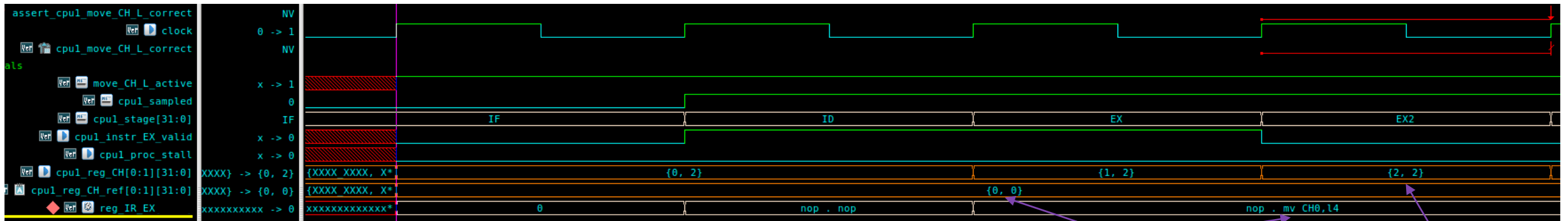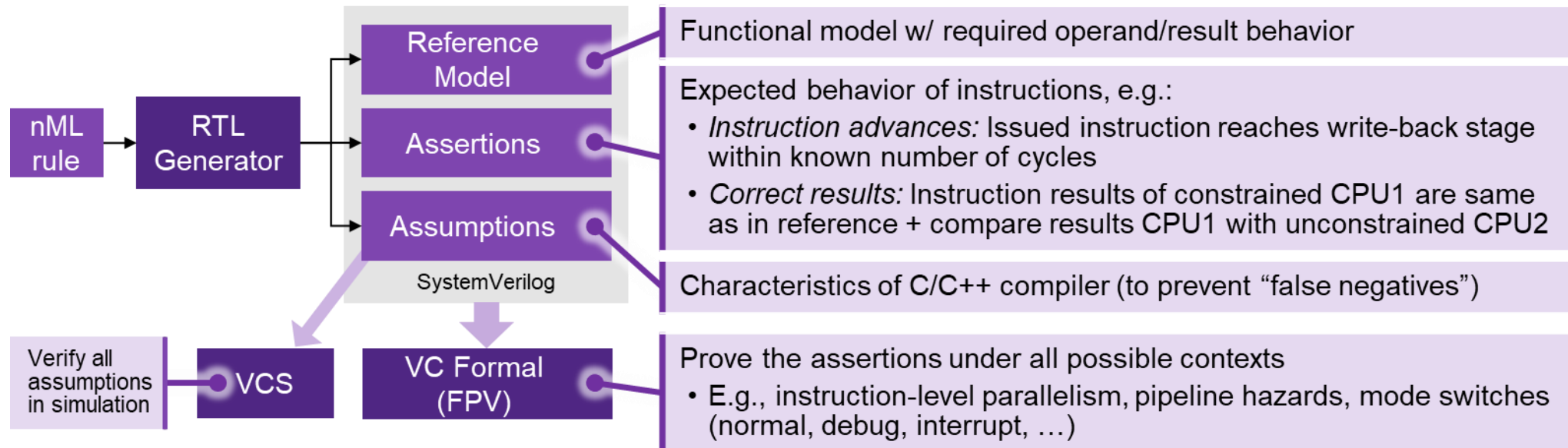
Reset of CH0 in EX
Value 0 expected in EX2

Wrong value 2
in EX2

Summary

- Automated formal ISA verification methodology for ASIP Designer

- Leveraging VC Formal FPV for processor verification (e.g. RISC-V)
  - Also applying other VC Formal apps (DPV, FTA, COV, …)
- Catches bugs that are hard to find by simulation