2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

MUNICH, GERMANY
OCTOBER 14-15, 2025

Out of The Box Techniques for Data-Path Verification

Atharva Mahesh Kakde

Ketki Gosavi, Pradeep Bagavathiappan, Anshul Singhal

cadence®

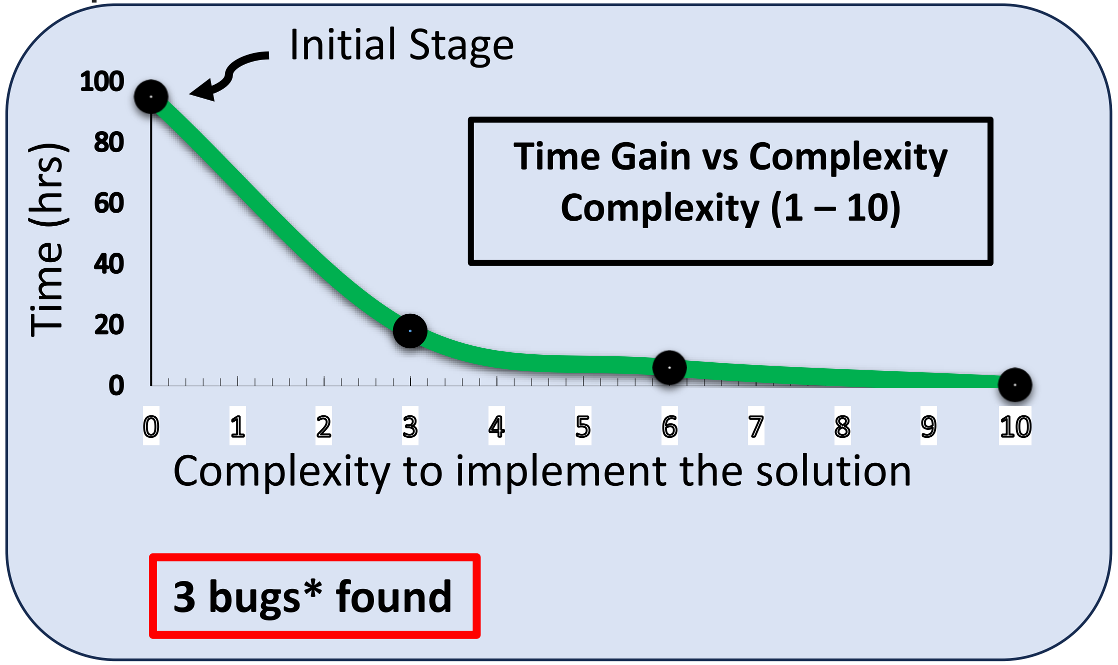
accellera SYSTEMS INITIATIVE

Agenda

- Introduction
- Design Details
- Flow used
- Challenges
- Proposed solutions
- Results
- Summary
- Q&A

Introduction

- **Challenge**
  - Verify C-vs-RTL equivalence for a complex data-path design – DUT (Image processing)
  - Difficult to compile original C++ model
  - Property non-convergence, once the compilation completes
- **Approach taken**
  - **Formal Model Generation**
    - Divide and Conquer
    - Symbolic Variable
    - Compile time over constraint
  - **Property Convergence**
    - Avoiding $past from the SVA expression
    - Arbitrary Miter Model
    - Assume Guarantee and Case Spilt



3 bugs* found

This presentation will showcase generic and scalable methods to deal with complex data-path designs.

Design Details

Image processing unit

- Lot of data transformations and **simulation(Dpi/Score boarding)** could never cover all possible input combinations
- **FFT unit** implemented a 2048-point FFT
  - With large data-in, butterfly, twiddle factors, scale
  - 8 point FFT operation as the base
  - Sine/Cosine lookup tables
  - Shift and complex multiplication at each stages
  - Previous stage output was fed to next stage – Pipeline design
  - Input stream 4K. 1 data sample per clock cycle
  - Iteration breakdown : 512x4x8 = 16384 ~ 16K
  - FFT point calculation : 512x2048 = 2048
- **Decompression**
  - Lzo-Decompression (Open Source https://github.com/torvalds/linux/blob/master/lib/lzo/lzo1x_decompress_safe.c)
  - Input – Compressed data-stream of variable size up to 4KB
  - Input data have set of instructions embedded in it
  - C++ model process the entire data in one go where as RTL takes multiple valid ready cycles

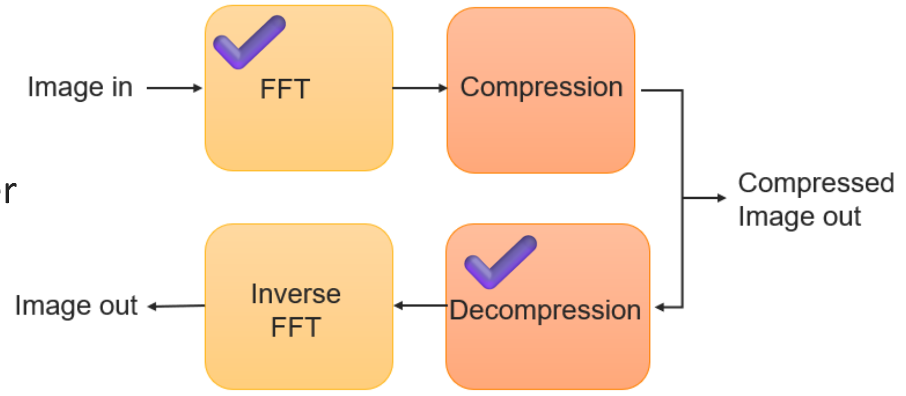
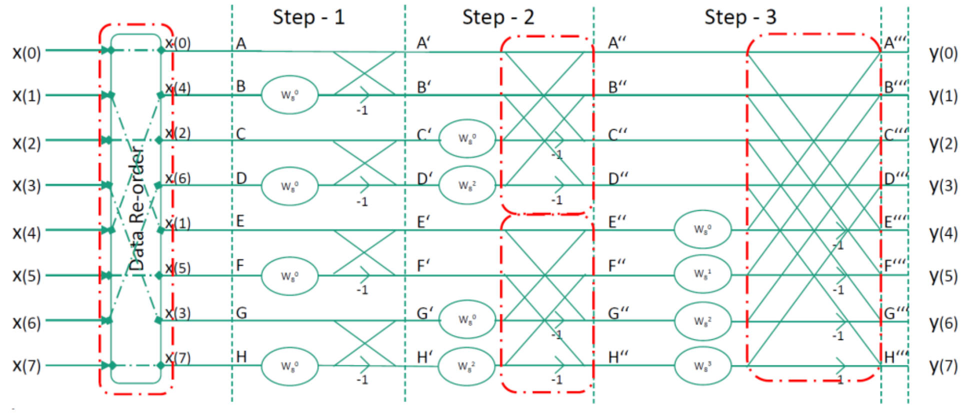
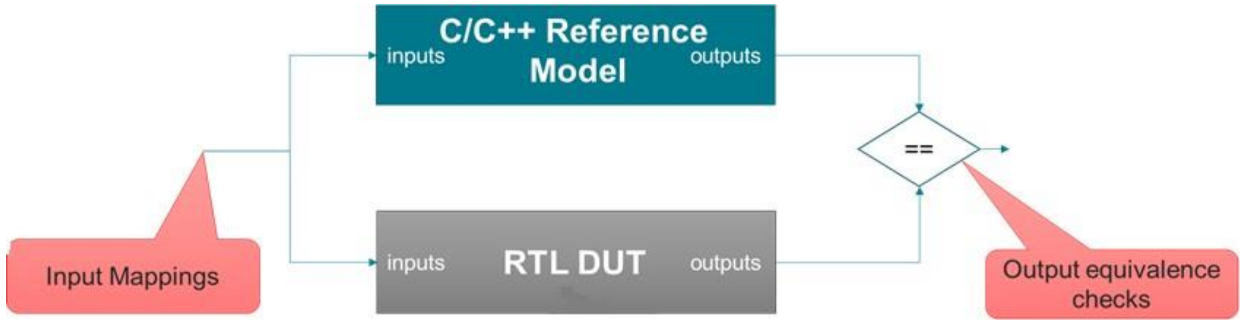


Image Processing Unit Block Diagram



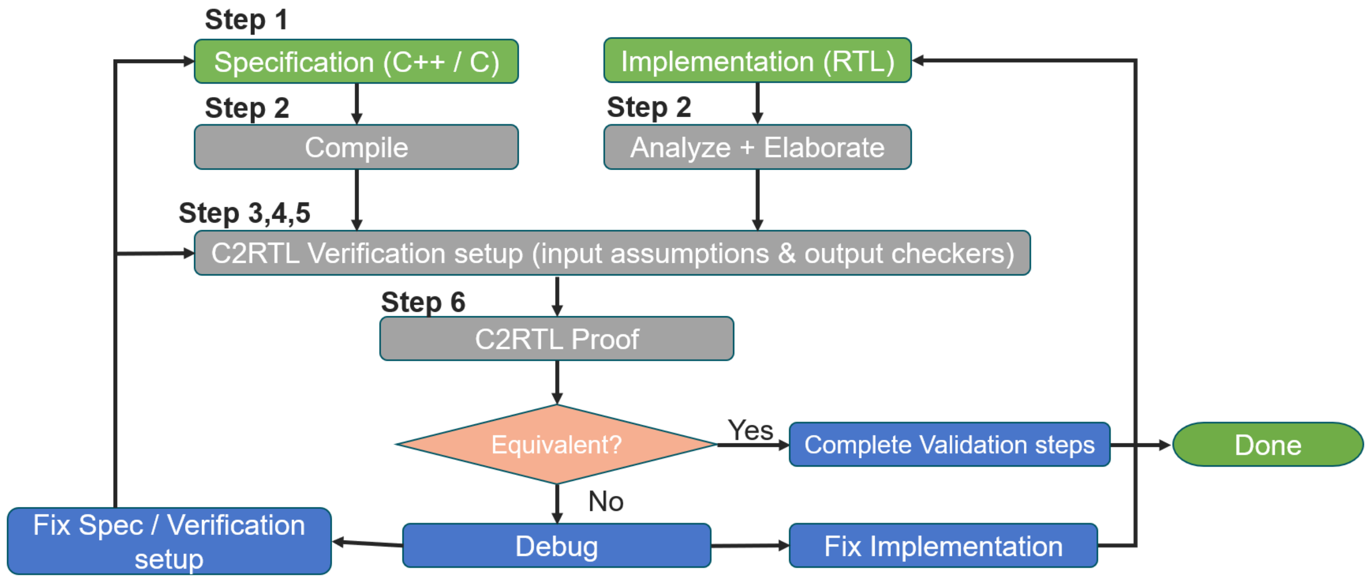
8 Point FFT Model

Flow Used



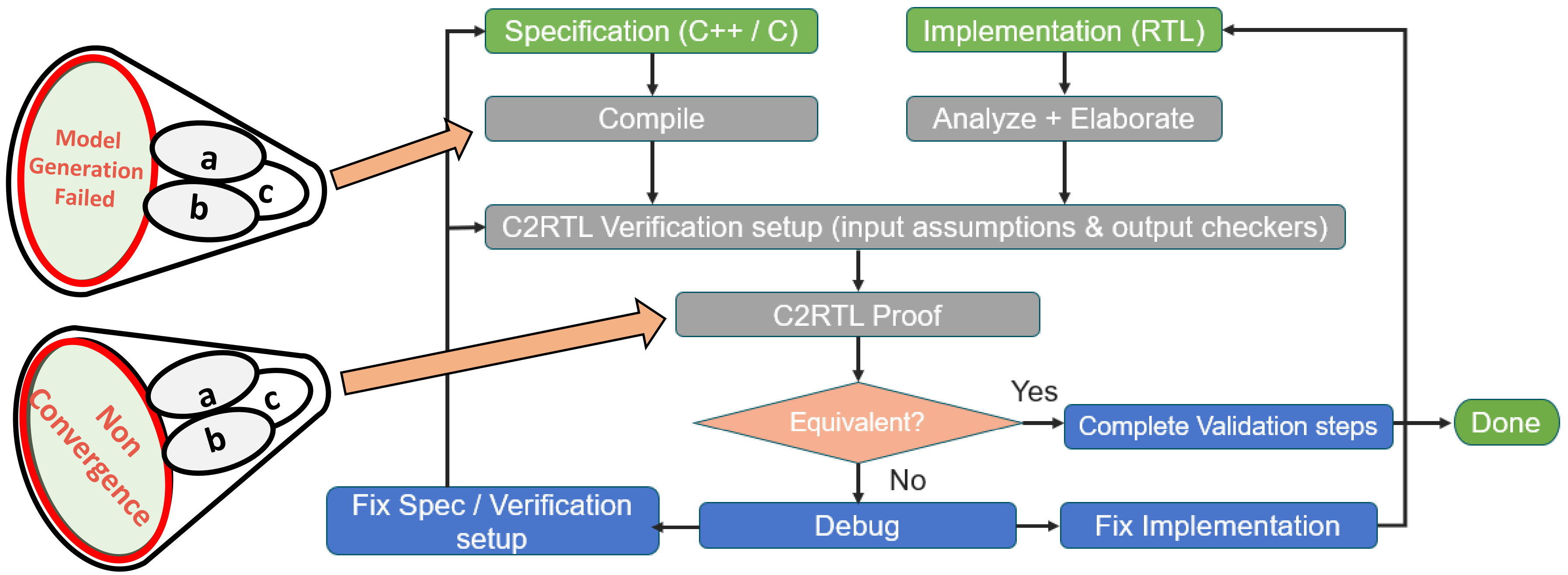
Miter Model

Equivalence Goal : For the same set of inputs to C/C++ Model and RTL, both are generating the same outputs, considering the pipeline delay



C2RTL Flow Diagram

Challenges



C++ model compile - Not able **to generate formal model** in reasonable time. **(>95 hrs)**

Solution a, b and c proposed in this presentation targets first step of C2RTL flow i.e. C++ model compile (Formal Model Generation)

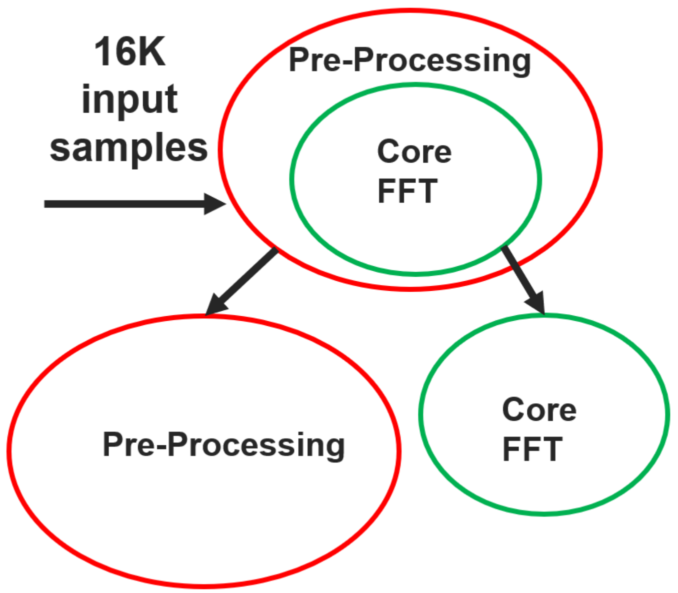
Proof non-convergence, once the formal model is generated

Solution a, b, and c address unique ways of handing convergence while doing C2RTL proof

*The above 2 challenges are show-stoppers, that would have rendered the verification completeness

1. Formal Model Generation
a) Divide and Conquer

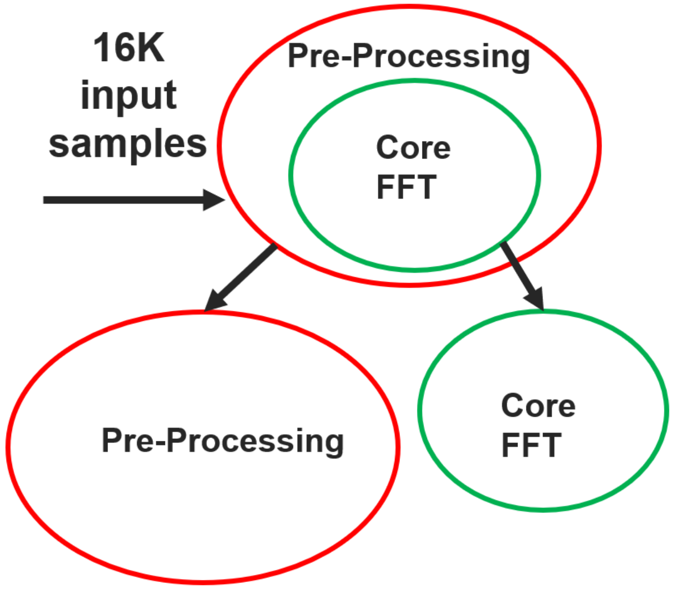
- The FFT unit consisted of 16K data samples pre-processing and was passed to the core FFT function
- C++ and RTL had separate function/module for core FFT functionality
- **Divided into 2 setups** – 1) Pre-Processing and 2) Core-FFT
- The **pre-processing setup equivalence** achieved between RTL and C++
- Core-FFT had loop iterating 2K times, to process 8-point FFT function call
  - ➢ It was a challenge to generate formal model -> **Symbolic Variable**



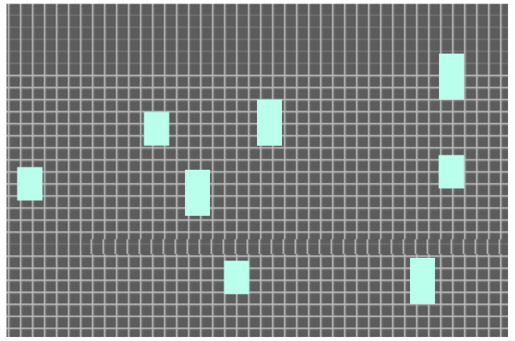
Verifying pre-processing and core FFT separately

1. Formal Model Generation
b) Symbolic variable

- Equivalence check passed with over-constrained input data to first 8 fixed reference values
- To represent these 8 data samples symbolically to any eight data samples out of 16K to cover all the scenarios
  - Assumptions were added in the C++ code to symbolically pick 8 twiddle and their corresponding scale value with random data
  - As these signals will now be driven as per the assumption, cutpoints were added to their corresponding RTL signal and assumed them to have same value as the C++ model.
  - Later we proved the assumptions separately
- This downsized the formal model, and it got generated in matter of seconds



Symbolic Variable for core FFT



1. Formal Model Generation
c) Compile time over constraint

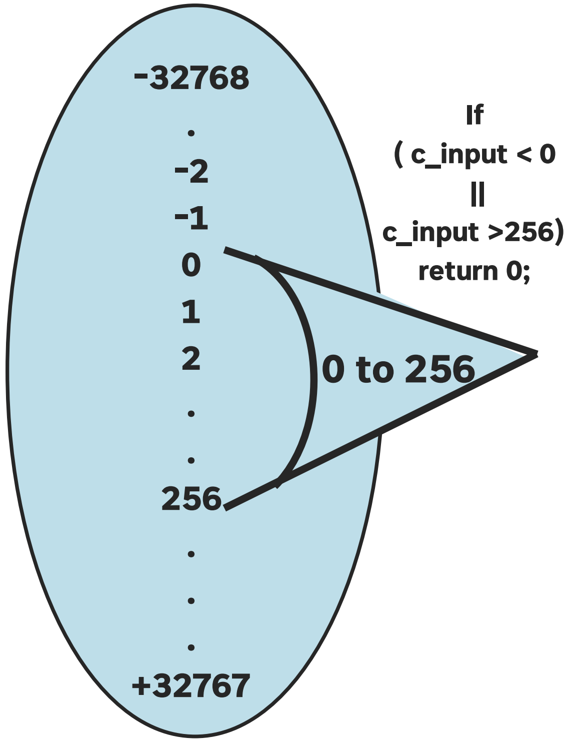
- Decompression unit input data stream consist of data+instructions
- A smart way to separate data from instructions

```
int16_t c_intput;
JASPER_INPUT(c_input);
```

`if(c_input < 0 || c_input > 256)return 0;`

The “return 0” idea !

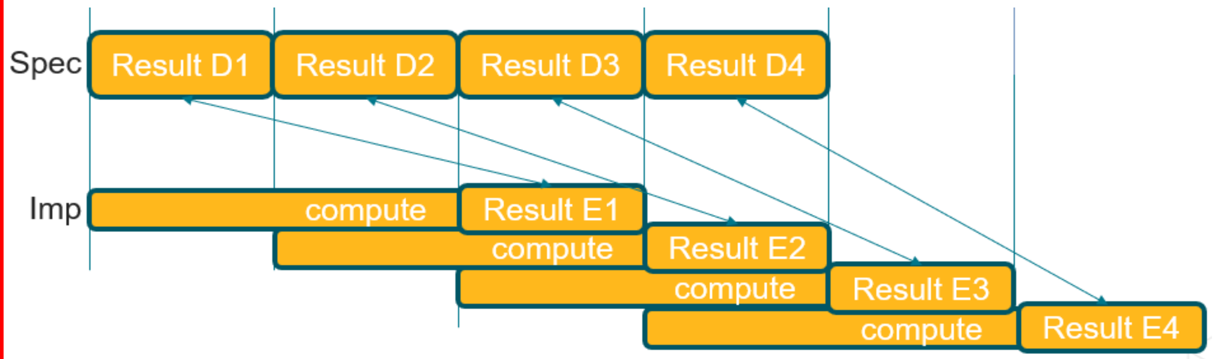
- `int16_t` is a 16-bit data type varying from `−32768 < 0 < 32767` i.e. formal model to be generated for `(2^16) = 65536` possible values on `c_input`
- Using `return 0` for all the values `<0 or >256` model generation will happen only for values `0 to 256`, which in intern reduces the size of C++ model
- Same way, multiple smaller models to be generated in separate setups, thereby getting a smaller netlist size



Compile time over constraint

get_design_info
Flops: 0 (0) (0 property flop bits)
Latches: 0 (0)
Gates: 1152209 (67300950)
Nets: 3523827

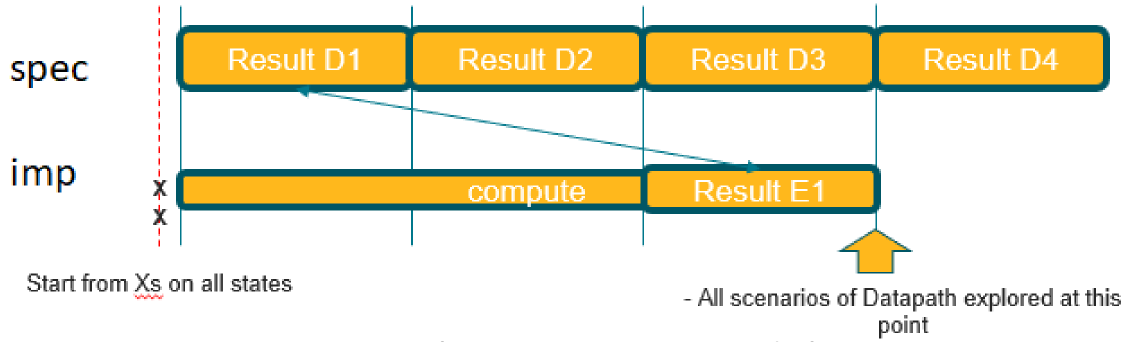
2. Convergence
a) Arbitrary miter model



Standard miter model

- The C++ model is untimed, but the decompression RTL required multiple cycles to initialize and exercise all relevant scenarios spread across time.
- In this case, multiple instances of the equivalence check need to be run as part of the standard miter model and this can make it difficult to converge.

- Instead, we kept the RTL uninitialized, to start from any arbitrary state, making it cover all the scenarios in only a single transaction.
- For this approach, we might require additional constraints at the RTL to avoid invalid scenarios


spec
Result D1
Result D2
Result D3
Result D4
imp
x
x
compute
Result E1
Start from Xs on all states
- All scenarios of Datapath explored at this point

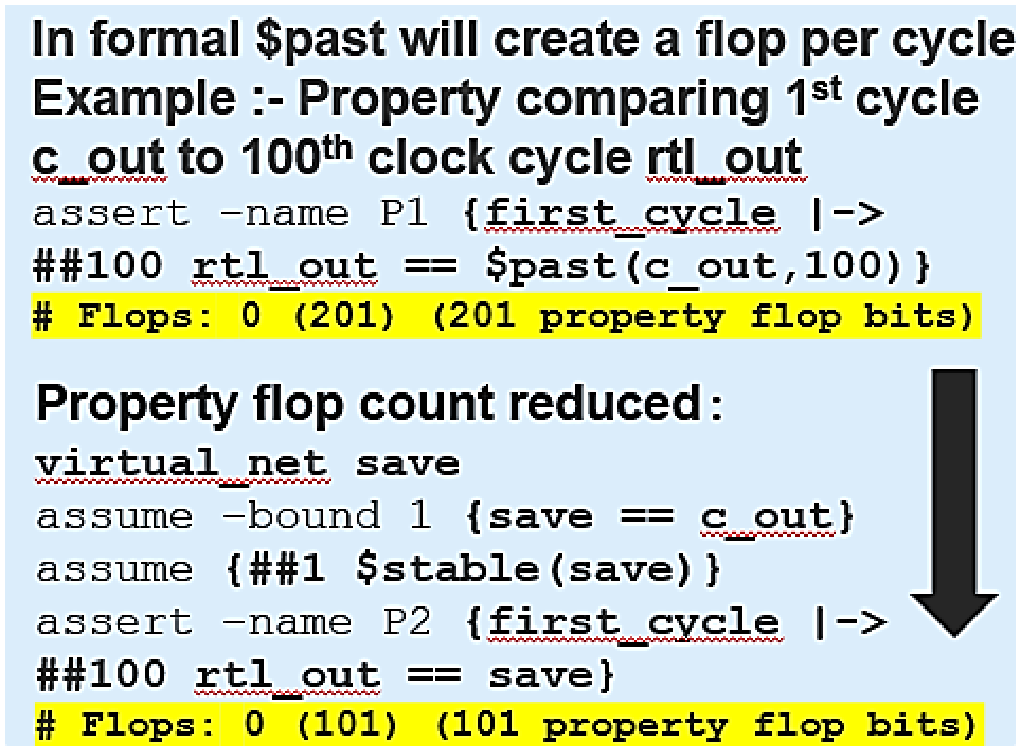
Arbitrary miter model

2. Convergence
b) Avoid the use of $past

- For CvsRTL equivalence check, it is often required to compare one cycle C++ output to Nth cycle RTL output due to sequential delays at RTL. For this requirement we use asserts like

```
assert {$past(c_out, N) == rtl_out}
```

- `$past` saves previous values of the signal in flops and then, evaluates at the required clock cycle.
- Higher the value of N, more the number of flops



In formal $past will create a flop per cycle
Example :- Property comparing 1st cycle c_out to 100th clock cycle rtl_out

```
assert -name P1 {first_cycle |->
##100 rtl_out == $past(c_out,100)}
# Flops: 0 (201) (201 property flop bits)
```

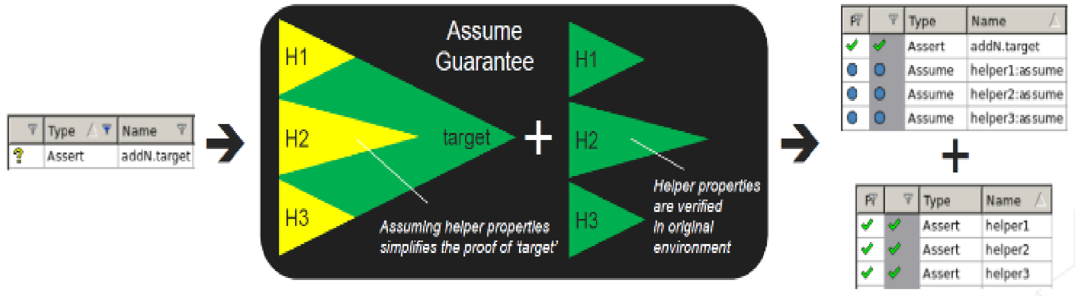
Property flop count reduced :

```
virtual_net save
assume -bound 1 {save == c_out}
assume {##1 $stable(save)}
assert -name P2 {first_cycle |->
##100 rtl_out == save}
# Flops: 0 (101) (101 property flop bits)
```

Avoid the use of $past

- **Smaller the flop count, faster the proof convergence**
- **Using Jasper specific tcl “virtual_net”, 1st cycle c_out is saved and compared to rtl_out in the 100th cycle**
- **Causing reduction in 100 property flops thereby reducing property complexity**

2. Convergence
c) Assume guarantee and Case split



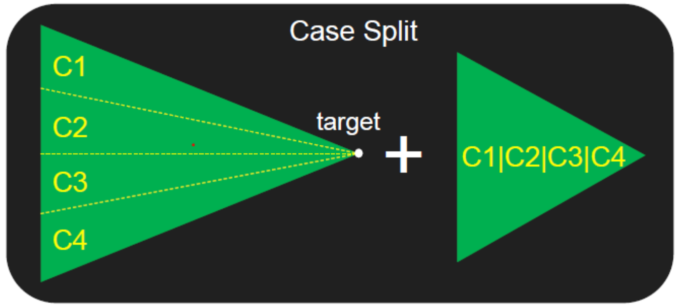
	Type	Name
?	Assert	addN.target

Assume Guarantee
H1
H2
H3
target
+
H1
H2
H3
Assuming helper properties simplifies the proof of 'target'
Helper properties are verified in original environment

Pr		Type	Name
✓	✓	Assert	addN.target
●	●	Assume	helper1:assume
●	●	Assume	helper2:assume
●	●	Assume	helper3:assume

Pr		Type	Name
✓	✓	Assert	helper1
✓	✓	Assert	helper2
✓	✓	Assert	helper3

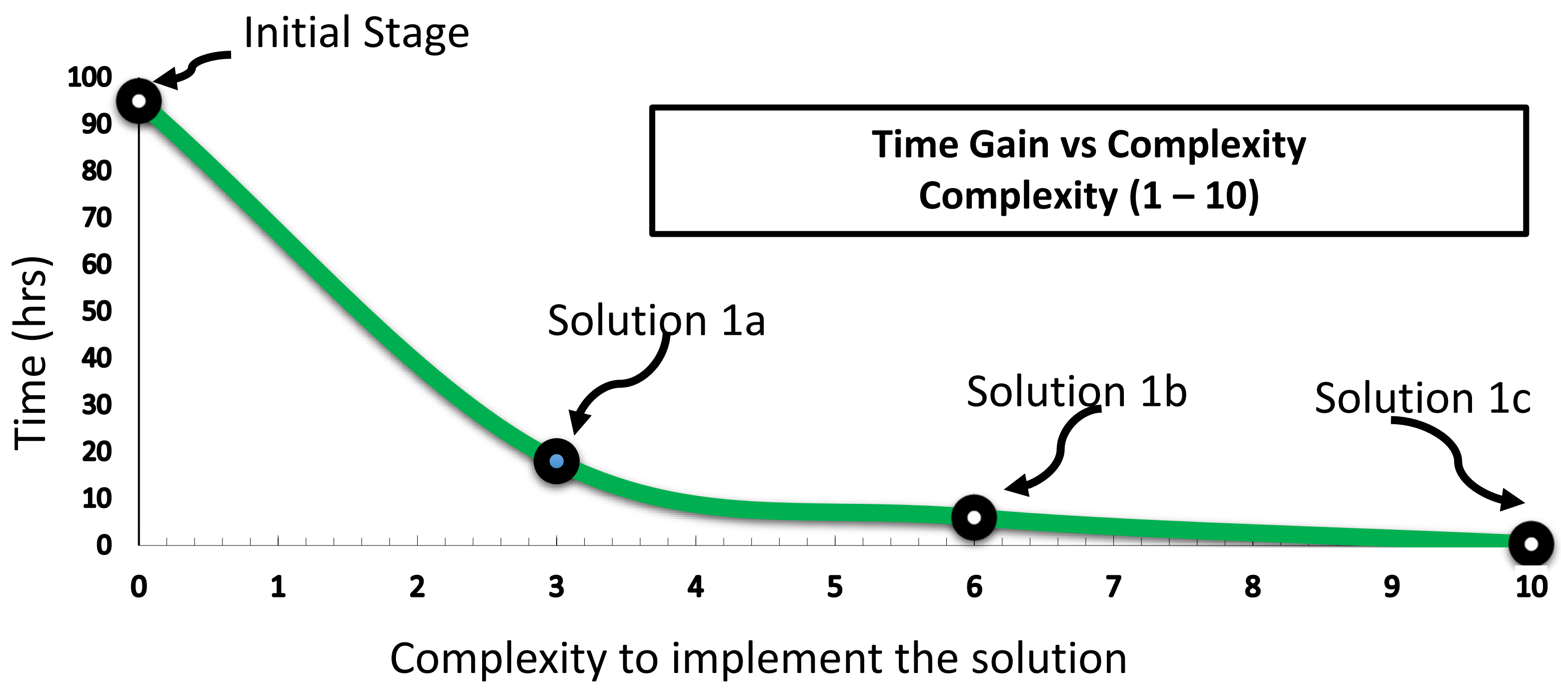
- Applied when end to end properties don't converge
- Best fit for pipelined design
- Intermediate proven properties act as assumptions for the top level target property
- Final stage of FFT (stage 4) equivalence checks were not converging
- So the stage 1 equivalence was proven and was assumed for proving stage 2 and so on stage 3 and 4.
- This helped in convergence as well as to locate stage wise complexity



- Applied when the property state space is huge, its difficult to target all the cases in COI
- Case split method applies case wise pre-condition to the target property
- Creates multiple splits of target properties with reduced number of scenarios to check and ensures it completeness
- Multiple cases were created in Decompression to target different combination of instructions separately

Results

- Compile time (Formal Model Generation time) brought down from >95hrs to <5mins



Detailed Results

Decompression

Parameter	Technique	Before application	After Application
Model Generation	**1.a**	Unable to generate	<5 Mins
Prove Time (Single Instruction)	**2.a** **2.b** **2.c**	>1day	<10mins
Prove Time (Multiple Instruction)		Not Converged	Max 2 days
Bug found	**2.a** **2.b**	Not converged	2 bugs* found in <1min

FFT

Parameter	Technique	Before Application	After Application
Model Generation	**1.b** **1.c**	Unable to generate	2mins
Prove time	**2.a** **2.b** **2.c**	Not Converged	Stage 1 - 113 sec Stage 2 - 88 Mins Stage 3 - 36 Mins Stage 4 - 119 Mins
Bugs found	**2.a** **2.b** **2.c**	Not converged	1Bug* found in <1min

***Bug Decompression** : FSM hang scenario
***Bug Decompression** : Non-equivalence due to unimplemented RTL scenario
***Bug FFT** : Non-equivalence due to stage wise tolerance difference in RTL and C++ model

Summary – I would like to leave you with this thought …………….

- **6 Generic techniques** for complex data-path designs
- **6 Independent techniques** for complex data-path designs
- Formal given its **exhaustive nature** - **Bug found** on pre-verified design
- Formal **does not need heavy testcase creation** like in dynamic simulation
- Even if we **don't achieve full convergence**, one can implement these techniques for
  - ✓ bug hunting
  - ✓ explore deeper bounds
  - ✓ negative testing by injecting bugs
- **Scalable:** Algorithms like **AES, SAM, FMUL etc**. were also proven using these techniques

Thank You ! ☺

Questions ?