Formal Verification for SystemC/C++ Designs

DVClub

September 2021

Vlada Kalinic, OneSpin: A Siemens Business

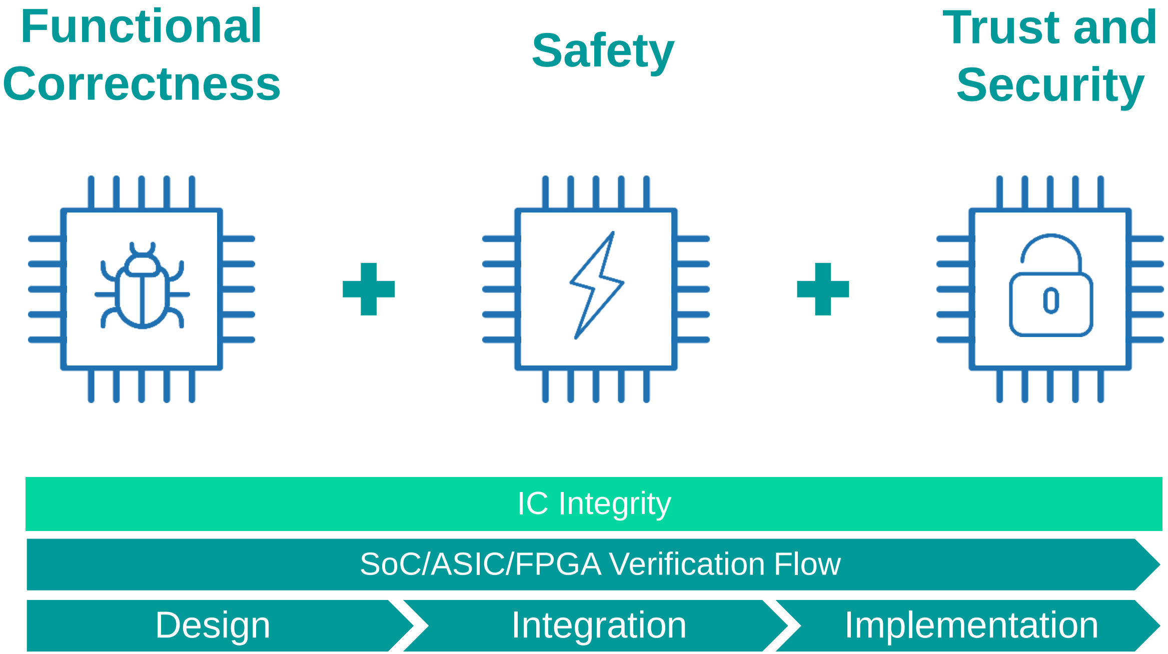
SIEMENS

Agenda

- Introductions
- Overview of HLS usage, current challenges, opportunities
- OneSpin – SystemC DV Inspect and Verify Overview
- Q&A

IC Integrity

Functionally Correct, Safe, Secure, and Trusted SoCs/ASICs/FPGAs



OneSpin: A Siemens Business provides certified **IC Integrity Verification Solutions** to develop functionally correct, safe, secure, and trusted integrated circuits.

Leading-Edge Formal Technology

Targeting Critical Hardware Verification Challenges

Functional Correctness

Rigorous coverage-driven functional verification from block to chip, leveraging formal technology

Design Exploration
Protocol Violations
Integrate Formal/Sim Coverage
End-to-End User Assertions
HLS/SystemC Verification
Synthesis/P&R Errors

Safety

Safety analysis and higher diagnostic coverage to meet strict certification requirements

FMEDA of Complex SoCs
Failure Mode Distribution
Avoid Excessive Fault Simulations
Measure Diagnostic Coverage
ISO 26262 Compliance
Tool Qualification

Trust and Security

Automated detection of RTL Trojans and hardware vulnerabilities to adversary attacks

Denial of Service
Data Leakage
Privileges Escalation
Data Integrity/Confidentiality
Hardware Backdoors
Hardware Trojans

OneSpin 360® Formal Platform

Heterogeneous Computing

Thorough verification of complex SoC platforms used for 5G wireless, IoT, and AI applications

Automotive and Industrial

Systematic bug elimination and metrics on proper handling of random errors in the field

RISC-V

RISC-V

Efficient and complete verification, including custom extensions. Compliance to ISA.

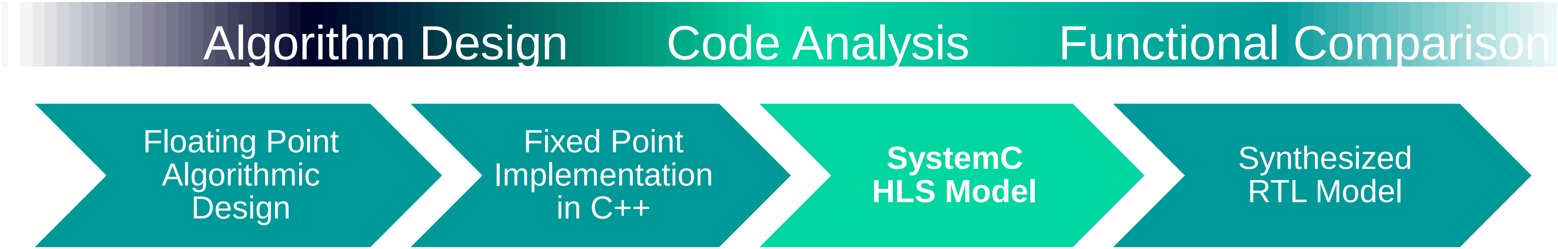
OneSpin Solutions and Services

SystemC with HLS Typical Issues

Imposing Hardware Constraints on C++

Using SystemC for HLS Modeling creates new problems and opportunities

- **Algorithm implementation issues in SystemC**
- **SystemC language related code problems**
- **Functional Consistency Checking of SystemC vs. RTL**



HLS Design Flow

SIEMENS

The Standards Do Not Help

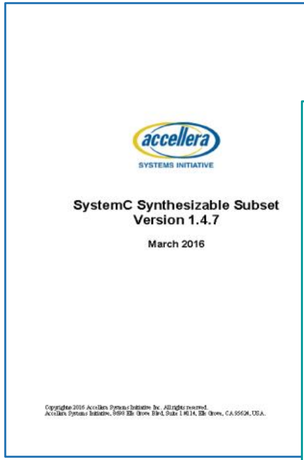
C++ not built for hardware descriptions

IEEE 1666 SystemC Standard

- 25+ occurrences of “unspecified“
- 50+ occurrences of “undefined“
- 150+ occurrences of “implementation defined“

Accellera Synthesizable Subset

- ~20 occurrences of “undefined“, “unspecified”, “implementation defined“



accellera
SYSTEMS INITIATIVE

SystemC Synthesizable Subset
Version 1.4.7

March 2016

Copyright© 2016 Accellera Systems Initiative Inc. All rights reserved.
Accellera Systems Initiative, 8698 Elk Grove Blvd, Suite 1 #014, Elk Grove, CA 95624, USA.

IEEE STANDARDS ASSOCIATION ◆IEEE

IEEE Standard for Standard SystemC® Language Reference Manual

IEEE Computer Society

Sponsored by the
Design Automation Standards Committee

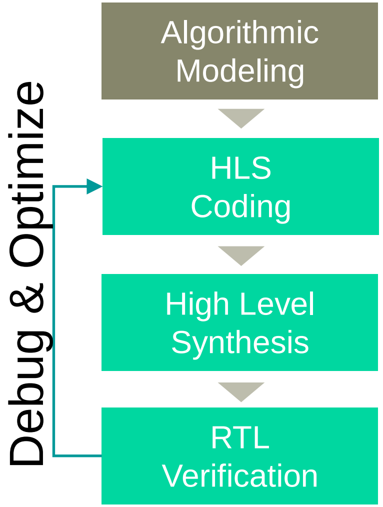
IEEE
3 Park Avenue
New York, NY 10016-5997
USA

9 January 2012

IEEE Std 1666™-2011
(Revision of
IEEE Std 1666-2005)

OneSpin: Advanced Verification for HLS

Current HLS Flow



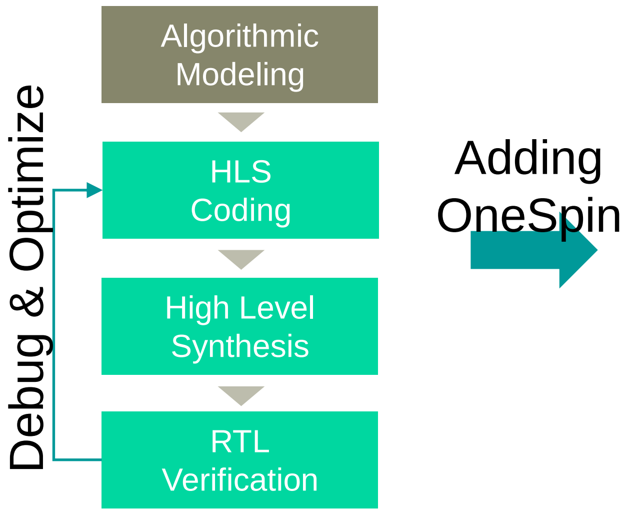
Challenges

- Limited useful feedback from HLS for coding style
- Certain coding mistakes can cause simulation mismatches that are extremely difficult to debug
- Optimization loop is long and somewhat ad-hoc
- Garbage in, garbage out

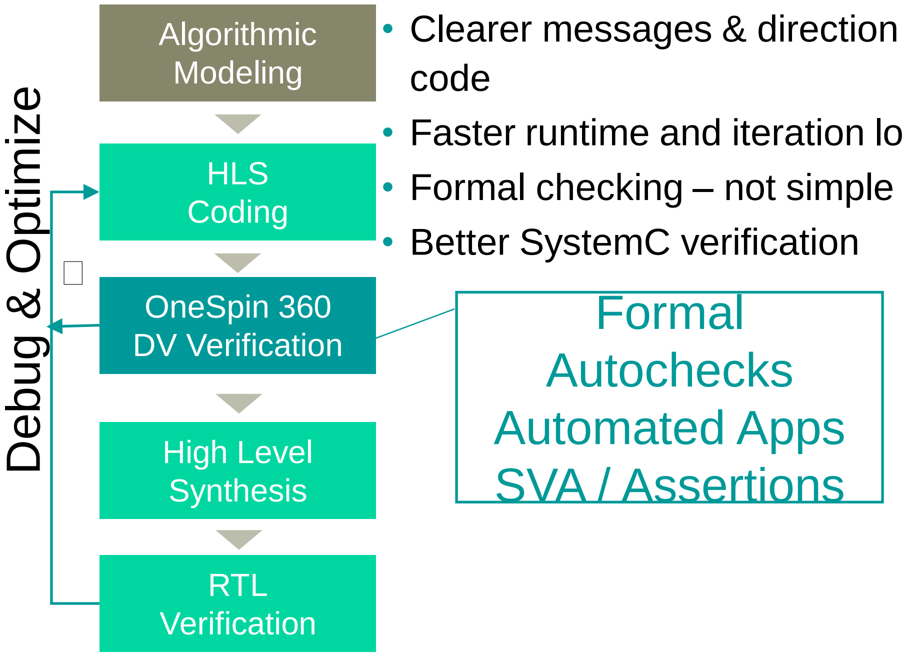
OneSpin: Advanced Verification for HLS

Opportunities to improve design flow

Current HLS Flow



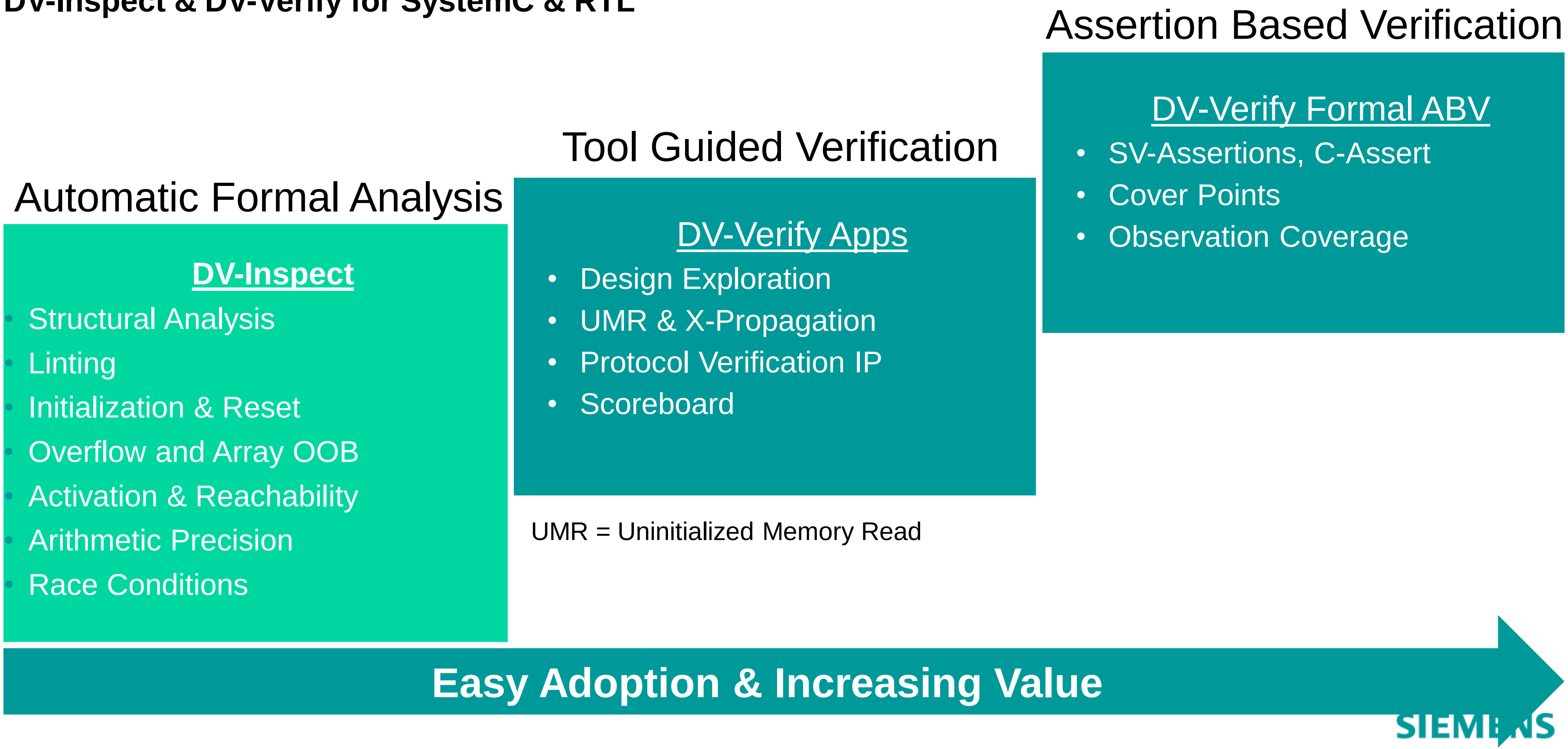
Improved HLS Flow



- Clearer messages & direction to improve code
- Faster runtime and iteration loop
- Formal checking – not simple linting
- Better SystemC verification

Deploying the OneSpin Products

DV-Inspect & DV-Verify for SystemC & RTL



Automatic Formal Analysis

DV-Inspect

- Structural Analysis
- Linting
- Initialization & Reset
- Overflow and Array OOB
- Activation & Reachability
- Arithmetic Precision
- Race Conditions

Tool Guided Verification

DV-Verify Apps

- Design Exploration
- UMR & X-Propagation
- Protocol Verification IP
- Scoreboard

UMR = Uninitialized Memory Read

Assertion Based Verification

DV-Verify Formal ABV

- SV-Assertions, C-Assert
- Cover Points
- Observation Coverage

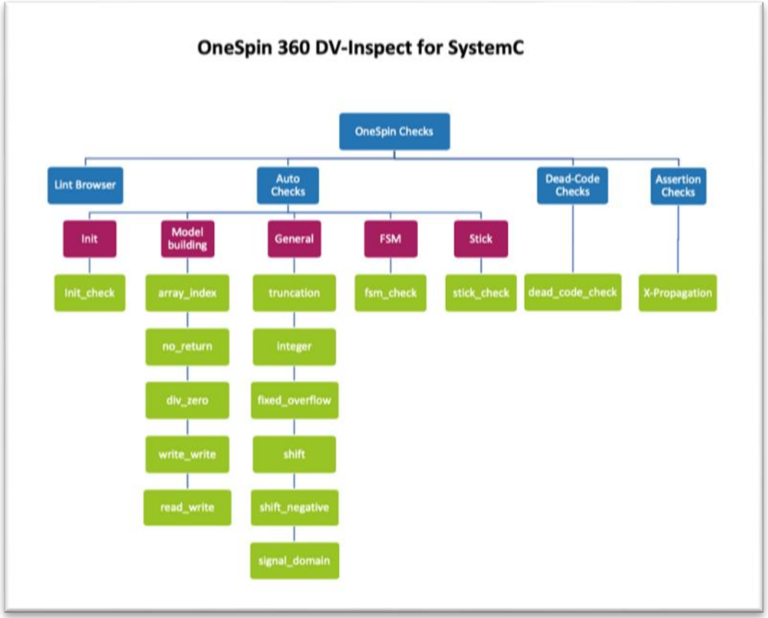
Easy Adoption & Increasing Value

OneSpin 360 DV High-Level Verification

Design Verification Solution for C++/SystemC HLS Code

Values

- Eliminate design bugs before HLS synthesis
- Start verification much earlier in the process
- Reduce simulation effort in SystemC and RTL
- Optimize HLS input code before synthesis



SystemC Inspect Automated Checks

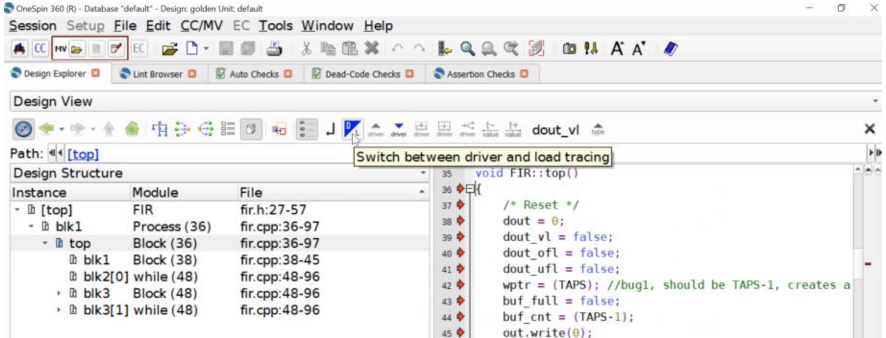
Tab	Autocheck	SystemC	Explanation
Init	init	X	Initialization checks are created for each non-redundant state signal and primary output of the current unit. An initialization check tests whether the corresponding signal is set to a uniquely determined value when applying the reset sequence of the unit.
ModelBuilding	array_index	X	An array index violation occurs if an array is accessed using an index which exceeds the array bounds. Array index checks check for static and dynamic violations in all array accesses occurring in the HDL source code.
ModelBuilding	div_zero	X	Division-By-Zero checks are generated for all arithmetic divisions occurring in Verilog. SystemC and VHDL source code, checking whether or not the divisor is always different from zero. These checks are also generated in Verilog and SystemC for modulo operations with a zero base and for pow operations on zero with a negative exponent.
ModelBuilding	no_return	X	Function-Without-Return checks test whether each possible control path through a function ends with a return statement.
General	shift_negative	2020.1	Checks whether a shift with a negative direction occurs. Cannot occur in SystemVerilog, since there shift counts are always treated as unsigned integers.
ModelBuilding	signal_domain	X	Signal domain checks investigate whether state bits of the unit can take a value other than zero or one, e.g. 'X' or 'Z'.
ModelBuilding	write_write	X	In Verilog and SystemC designs, it is possible that write-write races occur among different processes. In VHDL, a write-write race check is generated if a racing condition for a shared variable may occur.
ModelBuilding	read_write	2020.1	In Verilog and SystemC designs, it is possible that read-write races occur among different processes if blocking assignments are used. In VHDL, a read-write race check is generated if a racing condition for a shared variable may occur.

Tab	Autocheck	SystemC	Explanation
General	fixed_overflow	X	Checks for overflows in fixed_float implementations in VHDL and SystemC.
General	Integer	X	Integer checks are created for each signed or unsigned integer signal of the current unit. An integer check tests whether there are redundant bits in the signal.
General	shift	2020.1	A signal can be accidentally set to zero by logically shifting its value too many times in the same direction. For each shift operation occurring in the source code, a shift check is created, checking whether or not such unintended behavior may occur.
General	truncation	X	If the result of an integral operation is used in a context, that does not match the self-determined size or signedness of the operation, then relevant bits may be lost.
Dead Code	dead_code	X	A line of code is called dead code if it is not visited in any execution trace. Lines can be unreachable, for example, if the condition of an enclosing control structure never becomes true, thus always preventing it from being executed.
Stick	stick	X	Stick checks test the unit for constant bits in signals.
Assertion Checks	x_checking_setup x_checking	X	X-Propagation Analysis app provides a robust and effective circuit analysis that highlights all the issues in a design that could lead to X state propagations without reliance on simulation test stimulus.

Design Exploration

Design browser

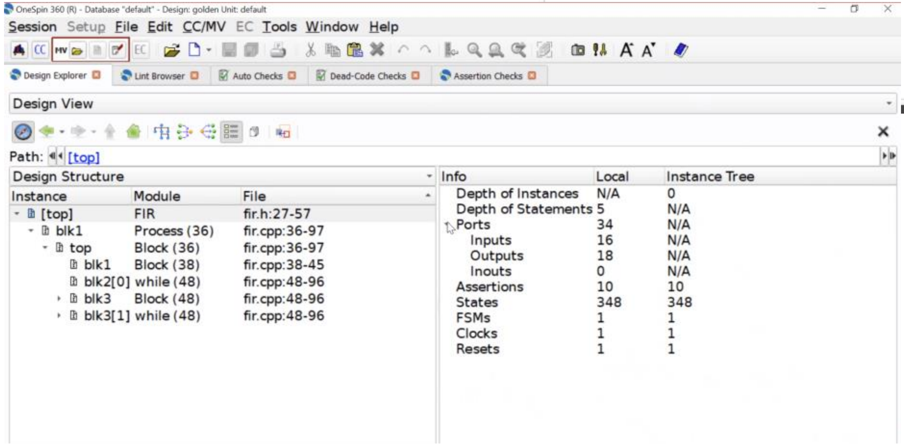
Full debugger


OneSpin 360 (R) - Database "default" - Design: golden Unit: default
Session Setup File Edit CC/MV EC Tools Window Help
Design Explorer
Lint Browser
Auto Checks
Dead-Code Checks
Assertion Checks
Design View
dout_vl
Switch between driver and load tracing
Path: [top]
Design Structure

Instance	Module	File
[top]	FIR	fir.h:27-57
blk1	Process (36)	fir.cpp:36-97
top	Block (36)	fir.cpp:36-97
blk1	Block (38)	fir.cpp:38-45
blk2[0]	while (48)	fir.cpp:48-96
blk3	Block (48)	fir.cpp:48-96
blk3[1]	while (48)	fir.cpp:48-96

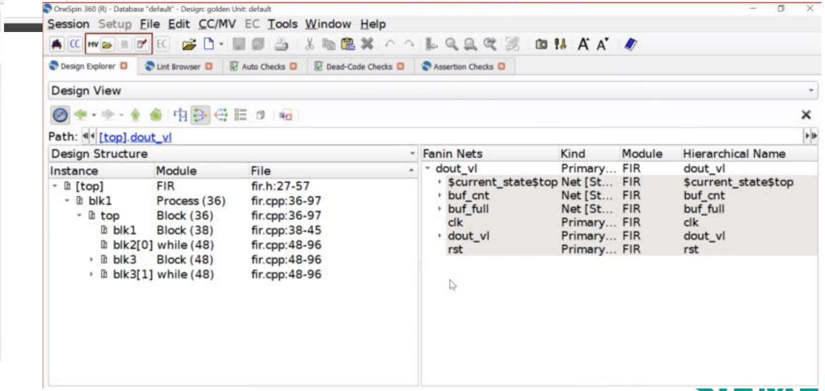
```
void FIR::top()
{
    /* Reset */
    dout = 0;
    dout_vl = false;
    dout_ofl = false;
    dout_ufl = false;
    wptr = (TAPS); //bug1, should be TAPS-1, creates a
    buf_full = false;
    buf_cnt = (TAPS-1);
    out.write(0);

    wait();
    while(1) {
       data_t delay_val;
```


OneSpin 360 (R) - Database "default" - Design: golden Unit: default
Session Setup File Edit CC/MV EC Tools Window Help
Design Explorer
Lint Browser
Auto Checks
Dead-Code Checks
Assertion Checks
Design View
Path: [top]
Design Structure

Instance	Module	File
[top]	FIR	fir.h:27-57
blk1	Process (36)	fir.cpp:36-97
top	Block (36)	fir.cpp:36-97
blk1	Block (38)	fir.cpp:38-45
blk2[0]	while (48)	fir.cpp:48-96
blk3	Block (48)	fir.cpp:48-96
blk3[1]	while (48)	fir.cpp:48-96

Info	Local	Instance Tree
Depth of Instances	N/A	0
Depth of Statements	5	N/A
Ports	34	N/A
Inputs	16	N/A
Outputs	18	N/A
Inouts	0	N/A
Assertions	10	10
States	348	348
FSMs	1	1
Clocks	1	1
Resets	1	1


OneSpin 360 (R) - Database "default" - Design: golden Unit: default
Session Setup File Edit CC/MV EC Tools Window Help
Design Explorer
Lint Browser
Auto Checks
Dead-Code Checks
Assertion Checks
Design View
Path: [top].dout_vl
Design Structure

Instance	Module	File
[top]	FIR	fir.h:27-57
blk1	Process (36)	fir.cpp:36-97
top	Block (36)	fir.cpp:36-97
blk1	Block (38)	fir.cpp:38-45
blk2[0]	while (48)	fir.cpp:48-96
blk3	Block (48)	fir.cpp:48-96
blk3[1]	while (48)	fir.cpp:48-96

Fanin Nets	Kind	Module	Hierarchical Name
dout_vl	Primary...	FIR	dout_vl
$current_state$top	Net [St...	FIR	$current_state$top
buf_cnt	Net [St...	FIR	buf_cnt
buf_full	Net [St...	FIR	buf_full
clk	Primary...	FIR	clk
dout_vl	Primary...	FIR	dout_vl
rst	Primary...	FIR	rst

Handling SystemC Initialization

Unpredictable reset states

Automatic variable initialization in SystemC (due to C++ mother language)

- All “sc_” datatypes automatically initialized to default value

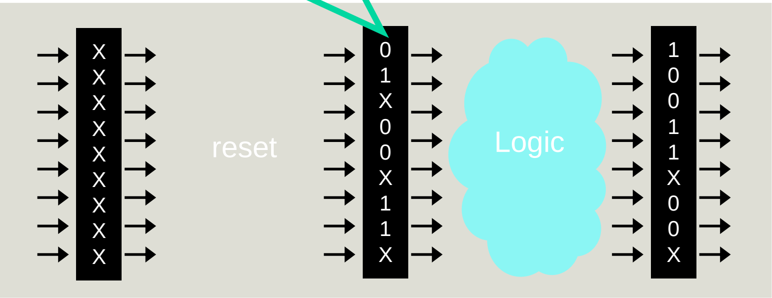
However, synthesizable subset standard states:

- Module constructor initializations ignored
- Reason: Reset behavior under user’s control

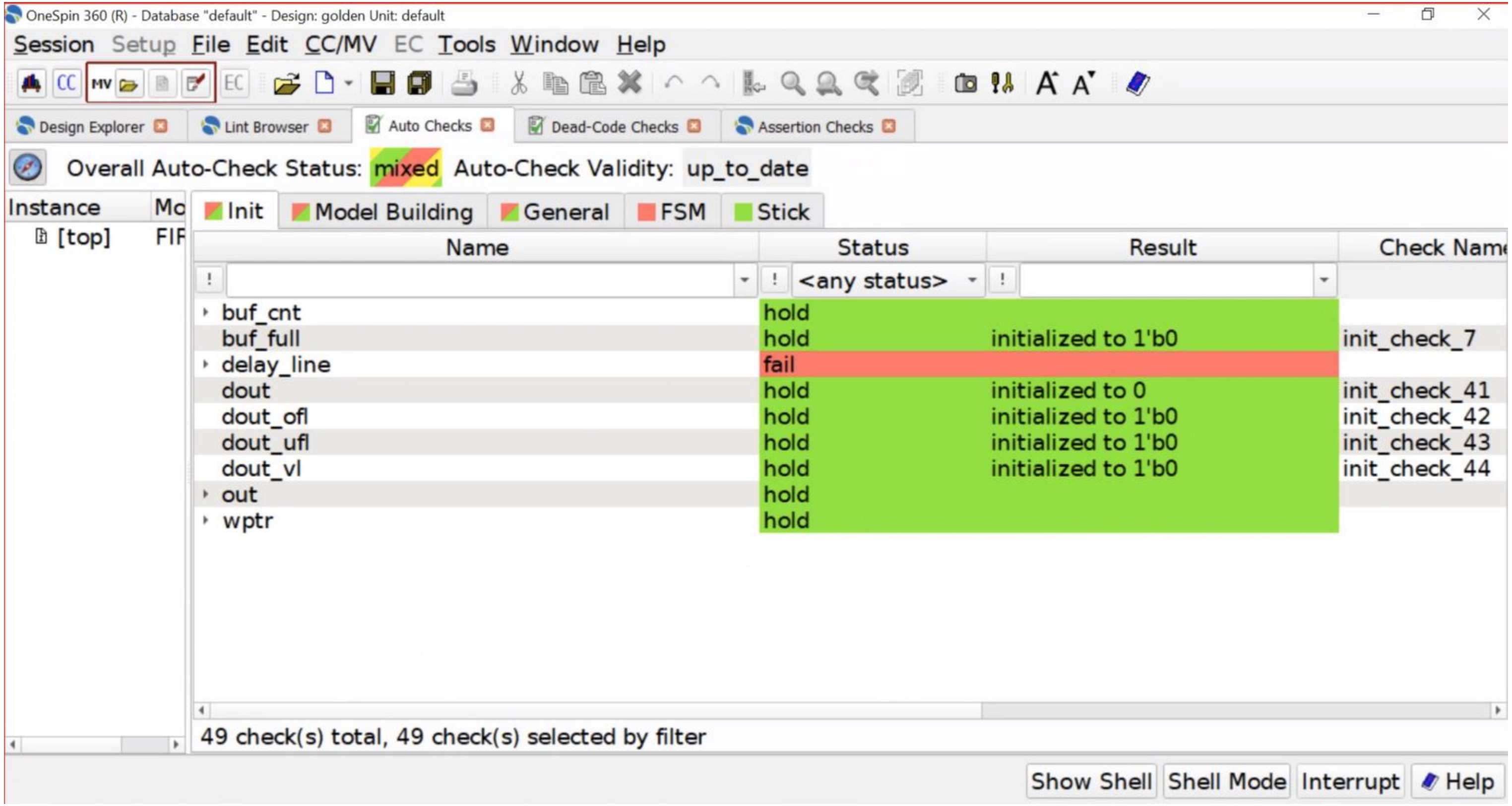
Inevitable Sim/Synth mismatches hard to debug using simulation

OneSpin 360 DV SystemC

✓ Checks which registers are initialized
✓ Check (intentionally) undefined reg effect
✓ Switch between sim & synth semantics



Initialization Checks



Overall Auto-Check Status: mixed Auto-Check Validity: up_to_date

Name	Status	Result	Check Name
buf_cnt	hold		
buf_full	hold	initialized to 1'b0	init_check_7
delay_line	fail		
dout	hold	initialized to 0	init_check_41
dout_ofl	hold	initialized to 1'b0	init_check_42
dout_ufl	hold	initialized to 1'b0	init_check_43
dout_vl	hold	initialized to 1'b0	init_check_44
out	hold		
wptr	hold		

49 check(s) total, 49 check(s) selected by filter

Undefined Value Propagation

No X-State in SystemC

Are all registers initialized?

- Uninitialized registers sources of X instability

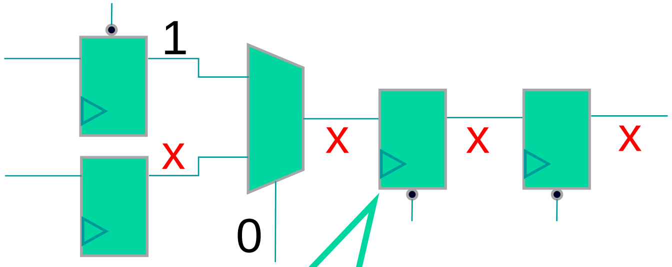
Other sources of X

- Undefined operations
- Multiple drivers

If Xs occur, will this have a bad effect?

Solutions?

- SystemC Simulator has no notion of undefined values or RTL semantics
- Formal can exhaustively analyze all conditions under which an X can propagate



OneSpin 360 DV SystemC

✓ Handles all sources of Xs

✓ Automated App to track X propagation

✓ Manual assertions (if Xs are allowed temporarily)

Undefined Operations

Example Array Out-Of-Bounds Access

- Simulation
  - Array address maybe larger than number of elements but no range checking
  - Undefined behavior with diverse effects
  - C++ checking tools slow and cumbersome
  - Trivial bugs are hard to find and debug

OneSpin 360 DV-Inspect

✓ Exhaustive analysis

✓ Precise error location

✓ Easy debug

```
sc_uint<8>  mem[8][16];
                  :
if(x>=16) { x = 15 };
if(y>=8) { y = 7 };
mem[x][y] = ...;
```

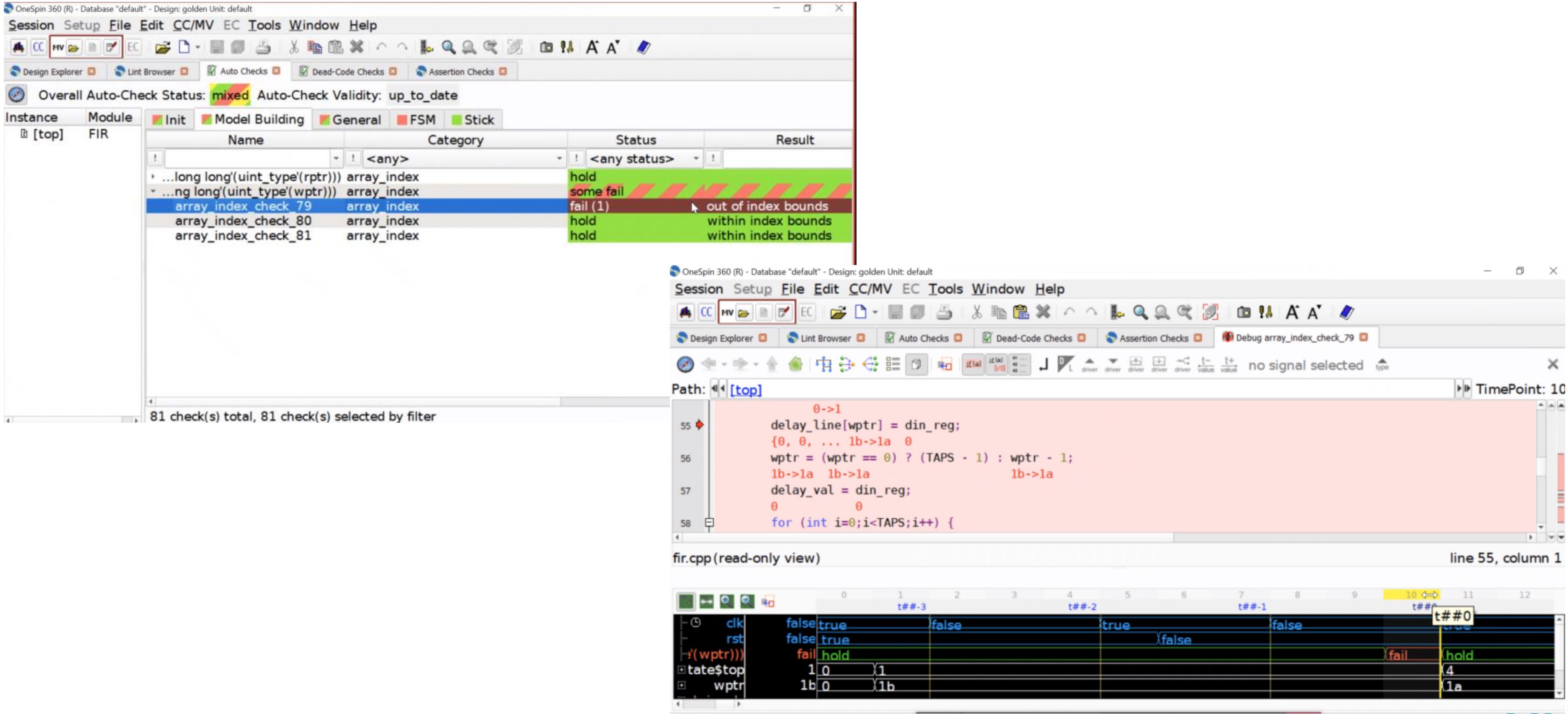
- Simulation does not complain and runs fine!
- DV-Inspect reports range violation error.

```
sc_signal<sc_uint<10> > intArr[10];
            :
int b = (large ? 10 : 5);
for(int i = 0; i <= b; ++i)
        intArr[i].write(0);
```

- Simulation does not complain but may crash if b = 10!
- DV-Inspect reports range violation error with b = 10 if ‚large‘ is possible.

SIEMENS

Array Out of Bounds



Overall Auto-Check Status: mixed Auto-Check Validity: up_to_date

Name	Category	Status	Result
!	! <any>	! <any status>	!
…long long'(uint_type'(rptr)))	array_index	hold	
…ng long'(uint_type'(wptr)))	array_index	some fail	
array_index_check_79	array_index	fail (1)	out of index bounds
array_index_check_80	array_index	hold	within index bounds
array_index_check_81	array_index	hold	within index bounds

81 check(s) total, 81 check(s) selected by filter

Path: [top] TimePoint: 10

```
          0->1
55      delay_line[wptr] = din_reg;
        {0, 0, ... 1b->1a  0
56      wptr = (wptr == 0) ? (TAPS - 1) : wptr - 1;
        1b->1a  1b->1a                1b->1a
57      delay_val = din_reg;
        0          0
58      for (int i=0;i<TAPS;i++) {
```

fir.cpp (read-only view) line 55, column 1

SIEMENS

SystemC Race Conditions

Simulation vs. Synthesis Mismatch

- **SystemC simulation is sequential**
  - Standard forces simulators to execute threads sequentially
  - No HDL-style “non-blocking” assignment

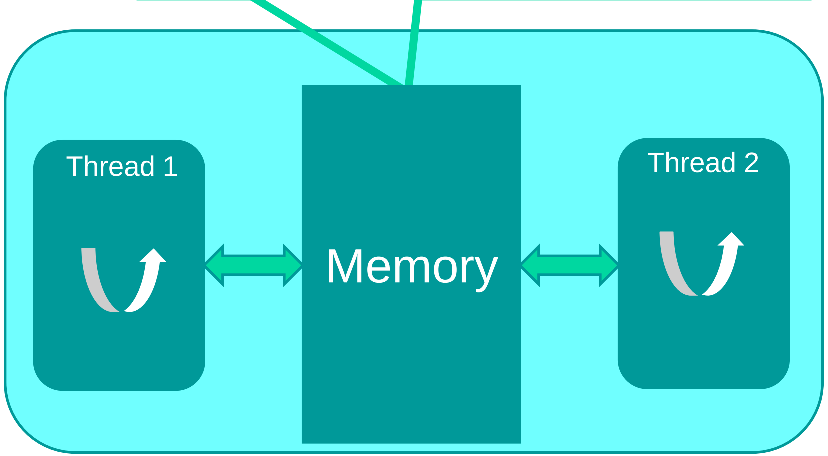
- **Hardware is concurrent**
  - RTL processes work in parallel
  - Synthesis result is parallel

- **HLS requires careful management of concurrent access to shared memories**

OneSpin 360 DV SystemC

✓ Implements synthesis semantics

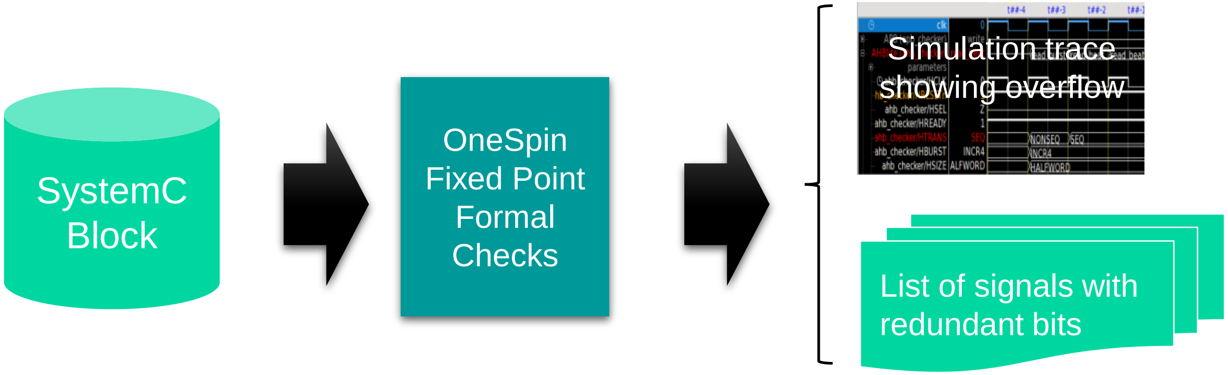
✓ Detects data races reliably



How does the designer guarantee no conflict?

Fixed Point Precision App

Automated redundancy and overflow checks


SystemC Block
OneSpin Fixed Point Formal Checks
Simulation trace showing overflow
List of signals with redundant bits

Check for overflow

- Check all operations for signed/unsigned overflow
- Full automation, no need for stimulus
- Prove absence of overflows
- Show traces of overflow scenarios

Check for redundant bits

- Checks uppermost bits for redundancy
- Automated, no need for stimulus
- Reports fixed point signals with redundant bits

Available for sc standard types and HLS

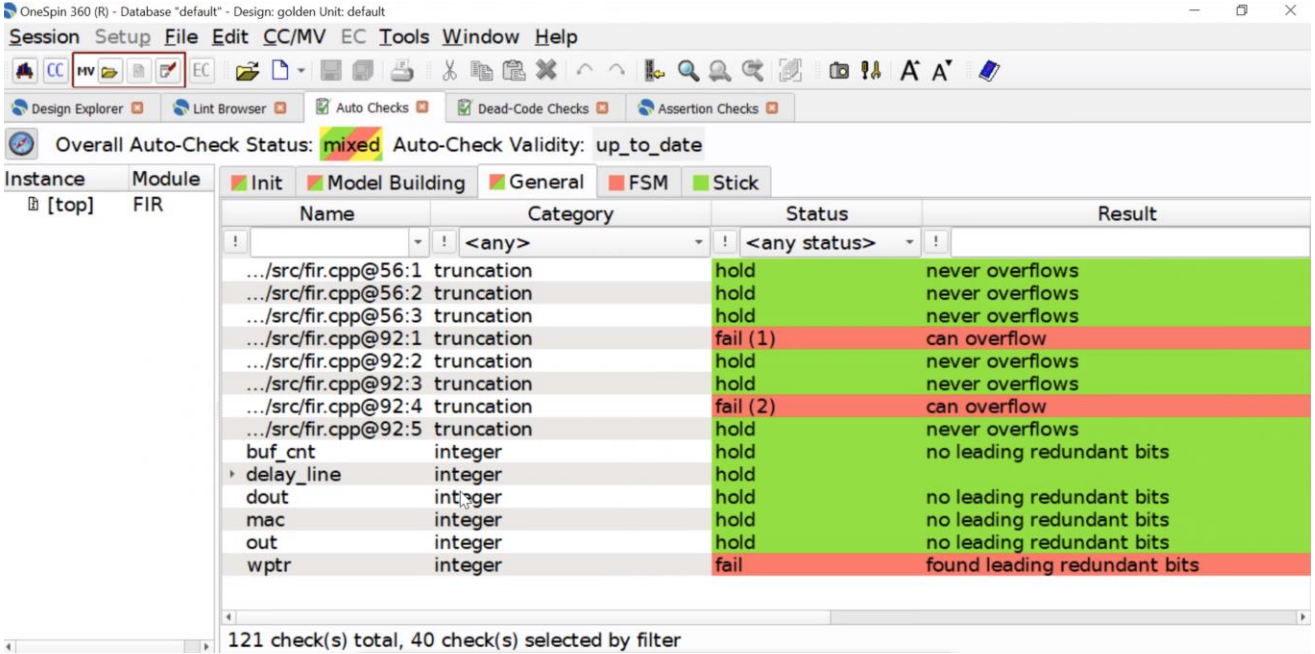
Finding Redundancy and Overflow

sc_(u)int

sc_fixed

int

cynw_(u)int

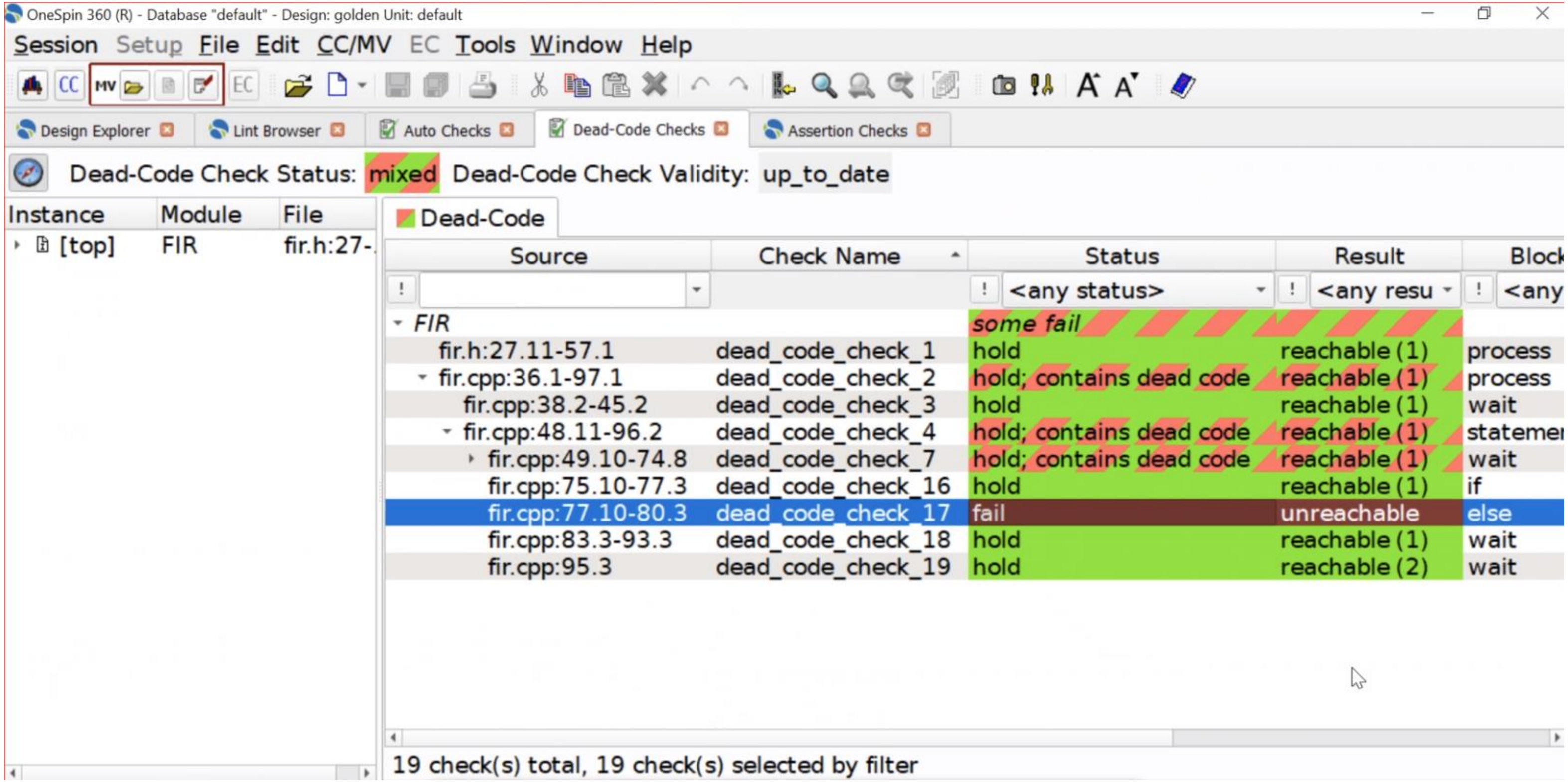


Overall Auto-Check Status: mixed Auto-Check Validity: up_to_date

Name	Category	Status	Result
.../src/fir.cpp@56:1	truncation	hold	never overflows
.../src/fir.cpp@56:2	truncation	hold	never overflows
.../src/fir.cpp@56:3	truncation	hold	never overflows
.../src/fir.cpp@92:1	truncation	fail (1)	can overflow
.../src/fir.cpp@92:2	truncation	hold	never overflows
.../src/fir.cpp@92:3	truncation	hold	never overflows
.../src/fir.cpp@92:4	truncation	fail (2)	can overflow
.../src/fir.cpp@92:5	truncation	hold	never overflows
buf_cnt	integer	hold	no leading redundant bits
delay_line	integer	hold	
dout	integer	hold	no leading redundant bits
mac	integer	hold	no leading redundant bits
out	integer	hold	no leading redundant bits
wptr	integer	fail	found leading redundant bits

121 check(s) total, 40 check(s) selected by filter

Dead Code Analysis


OneSpin 360 (R) - Database "default" - Design: golden Unit: default
Session Setup File Edit CC/MV EC Tools Window Help
CC
MV
EC
Design Explorer
Lint Browser
Auto Checks
Dead-Code Checks
Assertion Checks
Dead-Code

Dead-Code Check Status: mixed Dead-Code Check Validity: up_to_date

Instance	Module	File
[top]	FIR	fir.h:27-.

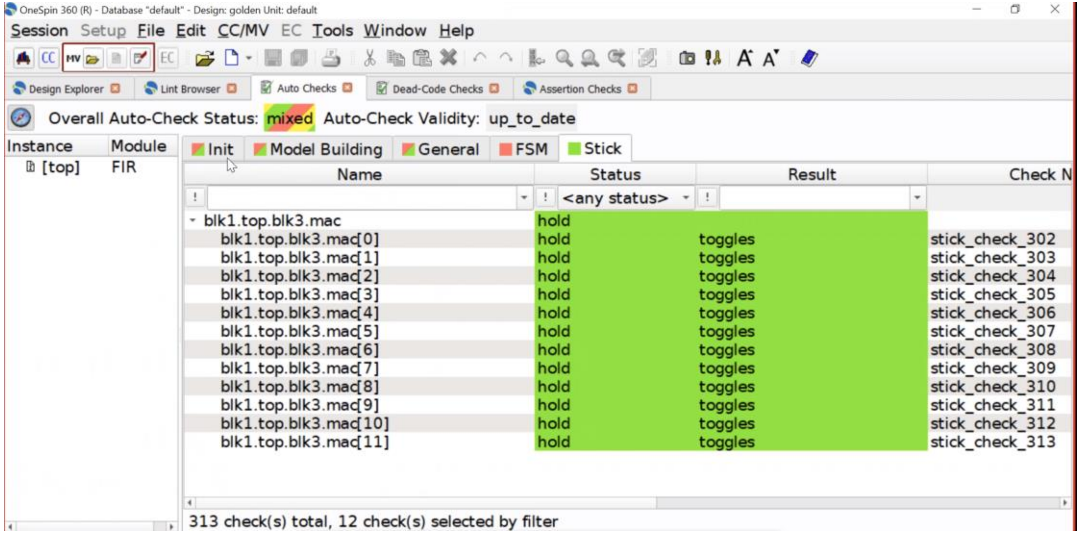
Source	Check Name	Status	Result	Block
!		! <any status>	! <any resu	! <any
FIR		some fail		
fir.h:27.11-57.1	dead_code_check_1	hold	reachable (1)	process
fir.cpp:36.1-97.1	dead_code_check_2	hold; contains dead code	reachable (1)	process
fir.cpp:38.2-45.2	dead_code_check_3	hold	reachable (1)	wait
fir.cpp:48.11-96.2	dead_code_check_4	hold; contains dead code	reachable (1)	statemer
fir.cpp:49.10-74.8	dead_code_check_7	hold; contains dead code	reachable (1)	wait
fir.cpp:75.10-77.3	dead_code_check_16	hold	reachable (1)	if
fir.cpp:77.10-80.3	dead_code_check_17	fail	unreachable	else
fir.cpp:83.3-93.3	dead_code_check_18	hold	reachable (1)	wait
fir.cpp:95.3	dead_code_check_19	hold	reachable (2)	wait

19 check(s) total, 19 check(s) selected by filter

Toggle Checks

“Stuck at” checks

Easily determines which bits are not used
or <u>not tested</u>!


OneSpin 360 (R) - Database "default" - Design: golden Unit: default
Session Setup File Edit CC/MV EC Tools Window Help
Design Explorer
Lint Browser
Auto Checks
Dead-Code Checks
Assertion Checks
Init
Model Building
General
FSM
Stick

Overall Auto-Check Status: mixed Auto-Check Validity: up_to_date

Instance	Module
[top]	FIR

Name	Status	Result	Check N
!	! <any status>	!	
blk1.top.blk3.mac	hold		
blk1.top.blk3.mac[0]	hold	toggles	stick_check_302
blk1.top.blk3.mac[1]	hold	toggles	stick_check_303
blk1.top.blk3.mac[2]	hold	toggles	stick_check_304
blk1.top.blk3.mac[3]	hold	toggles	stick_check_305
blk1.top.blk3.mac[4]	hold	toggles	stick_check_306
blk1.top.blk3.mac[5]	hold	toggles	stick_check_307
blk1.top.blk3.mac[6]	hold	toggles	stick_check_308
blk1.top.blk3.mac[7]	hold	toggles	stick_check_309
blk1.top.blk3.mac[8]	hold	toggles	stick_check_310
blk1.top.blk3.mac[9]	hold	toggles	stick_check_311
blk1.top.blk3.mac[10]	hold	toggles	stick_check_312
blk1.top.blk3.mac[11]	hold	toggles	stick_check_313

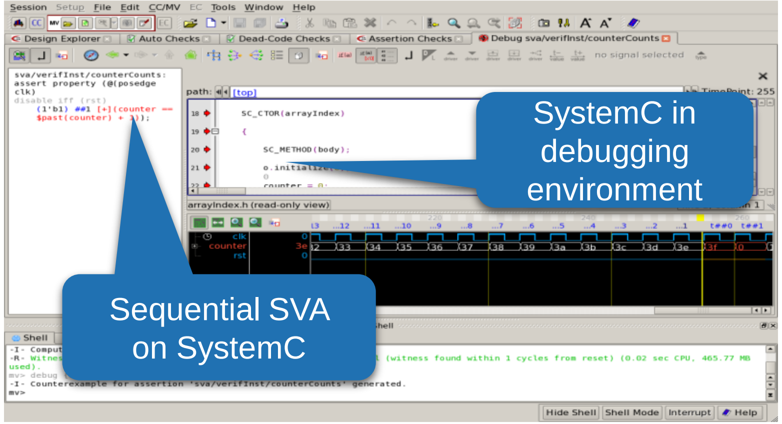
313 check(s) total, 12 check(s) selected by filter

Other Capabilities

SystemC Property Checking Solution

Leveraging SVA on SystemC

- Test specification elements against algorithm
- Consistent SystemVerilog assertions pre- and post-synthesis



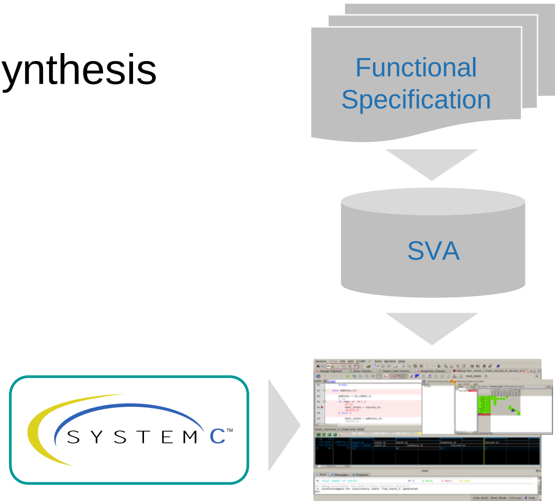
```
sva/verifInst/counterCounts:
assert property (@(posedge
clk)
disable iff (rst)
    (1'b1) ##1 [+](counter ==
    $past(counter) + 1));
```

```
    SC_CTOR(arrayIndex)
    {
        SC_METHOD(body);
        o.initialize(
        0
        counter = 0;
```

```
-I- Comput
-R- Witnes                    l (witness found within 1 cycles from reset) (0.02 sec CPU, 465.77 MB
used).
mv> debug
-I- Counterexample for assertion 'sva/verifInst/counterCounts' generated.
mv>
```

SystemC in debugging environment

Sequential SVA on SystemC



C++/SystemC Code

Formal Tool

Assertion Based Verification

Assertion classification

Type	assert	assume	cover
Description	Assertion	Constraint	Cover point
Purpose	Monitor DUT behavior	“Monitor” DUT inputs	Collect coverage data
Simulation	Eliminate ‘fail’ from TBs		Achieve ‘pass’ in TBs
Formal	Ensure absence of ‘fail’ by proving assertion	Assume absence of ‘fail’ (never show trace where assume fails)	Automatically find ‘pass’ or prove absence of ‘pass’

Distinction between assert/assume only important for formal

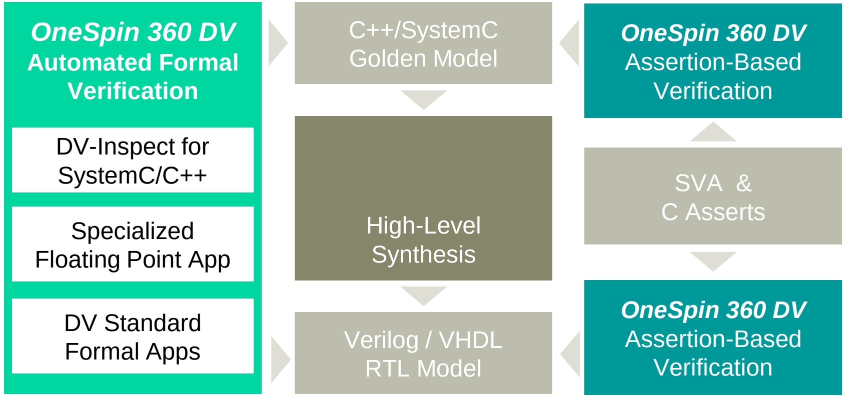
Formal typically requires assumes in order to avoid unrealistic fails for asserts

OneSpin with HLS Partnership

Design Verification Solution for HLS tools

- Cooperation with HLS teams
- Support of HLS libraries and coding
- Provides ***independent*** check on HLS flow

More efficient analysis and debug of C++/SystemC model prior high-level synthesis



Re-Use of assertions and apps on RTL for consistency

Use formal first! Improves verification flow! Gets working RTL faster!

OneSpin SystemC/C++ Solution

Enabling the HLS flow

SystemC/C++ Hardware Verification

- Currently tools do not address verification challenges
- HLS driving need for pre-synthesis verification

Language and Algorithm Verification Needs

- SystemC artifacts cause problems downstream
- Algorithm verification can be accelerated with automation

OneSpin: Unique SystemC Formal Solution

- Automation to significantly improve SystemC testing
- SystemVerilog assertions for flow continuity

For more information please visit

www.onespin.com

Thank you!