Using DPV for code coverage closure optimization

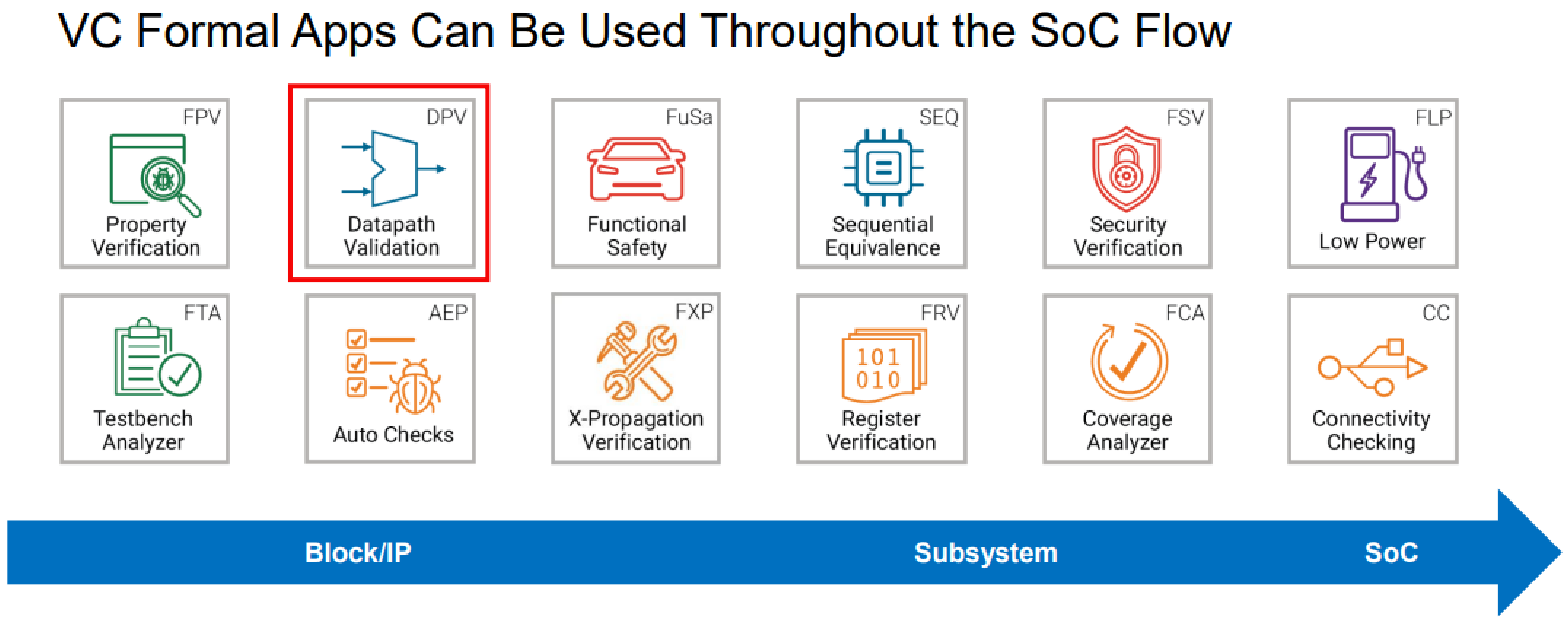
Kees van Kaam, verification lead
Dreamchip Technologies

Outline

- DPV introduction
- Verification context
- Closing code coverage of arithmetic modules
- DPV flow details
- Example
- Conclusion

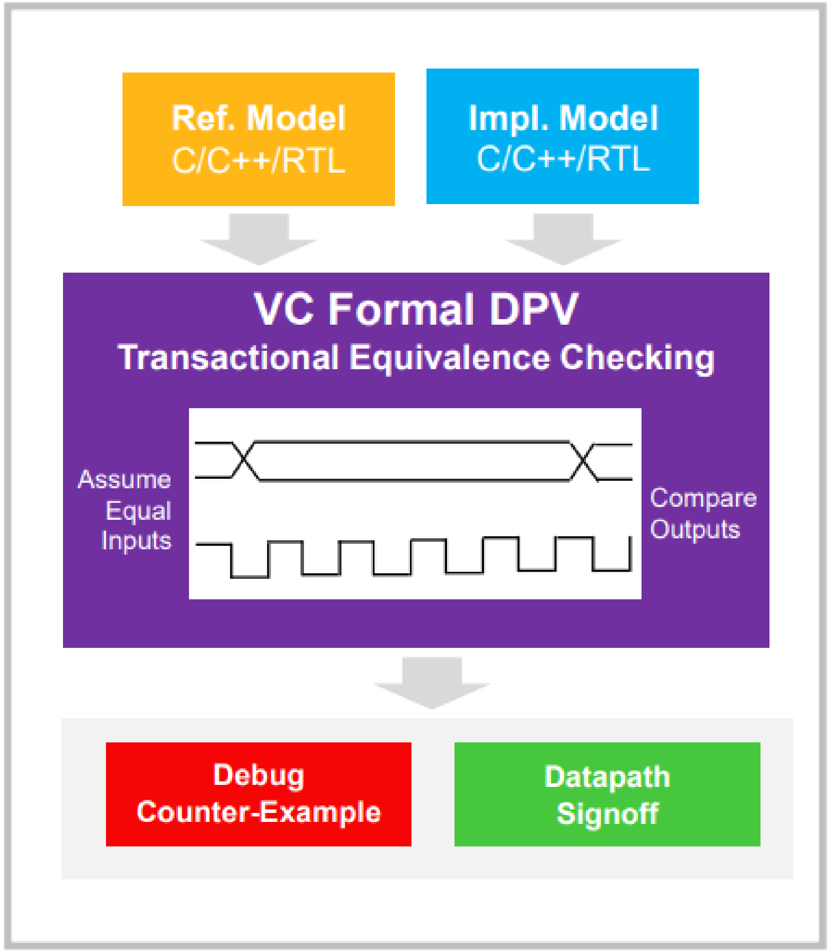
Introduction to DPV

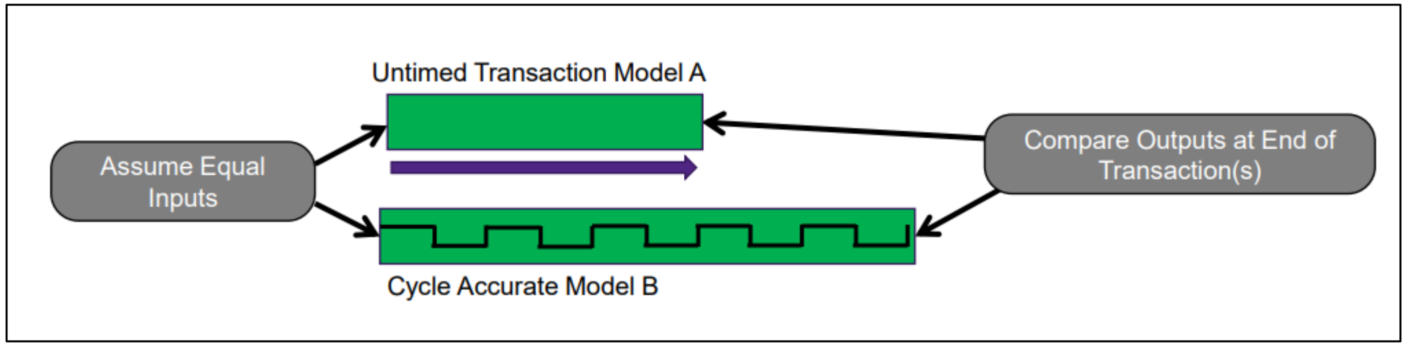
Synopsys VC formal DPV app



source: Synopsys

DPV equivalence checking




Untimed Transaction Model A
Assume Equal Inputs
Compare Outputs at End of Transaction(s)
Cycle Accurate Model B

Using equivalence checking using formal verification.
- Build in math functions for performance
- Uses the VC formal tool environment
- Reference model and implementation languages can be C/C++ or RTL

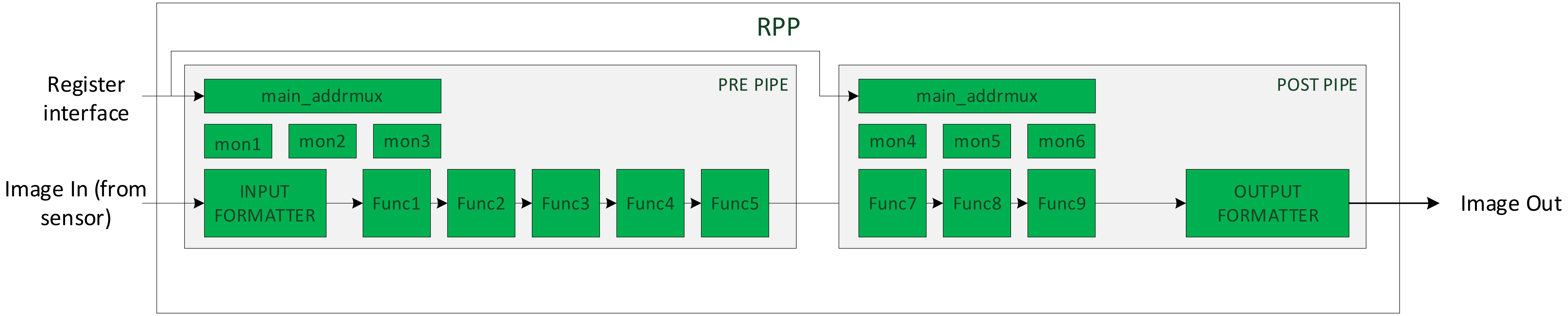
picture source: Synopsys

Verification context

Realtime Pixel Processor module verification

Realtime Pixel processor (RPP)

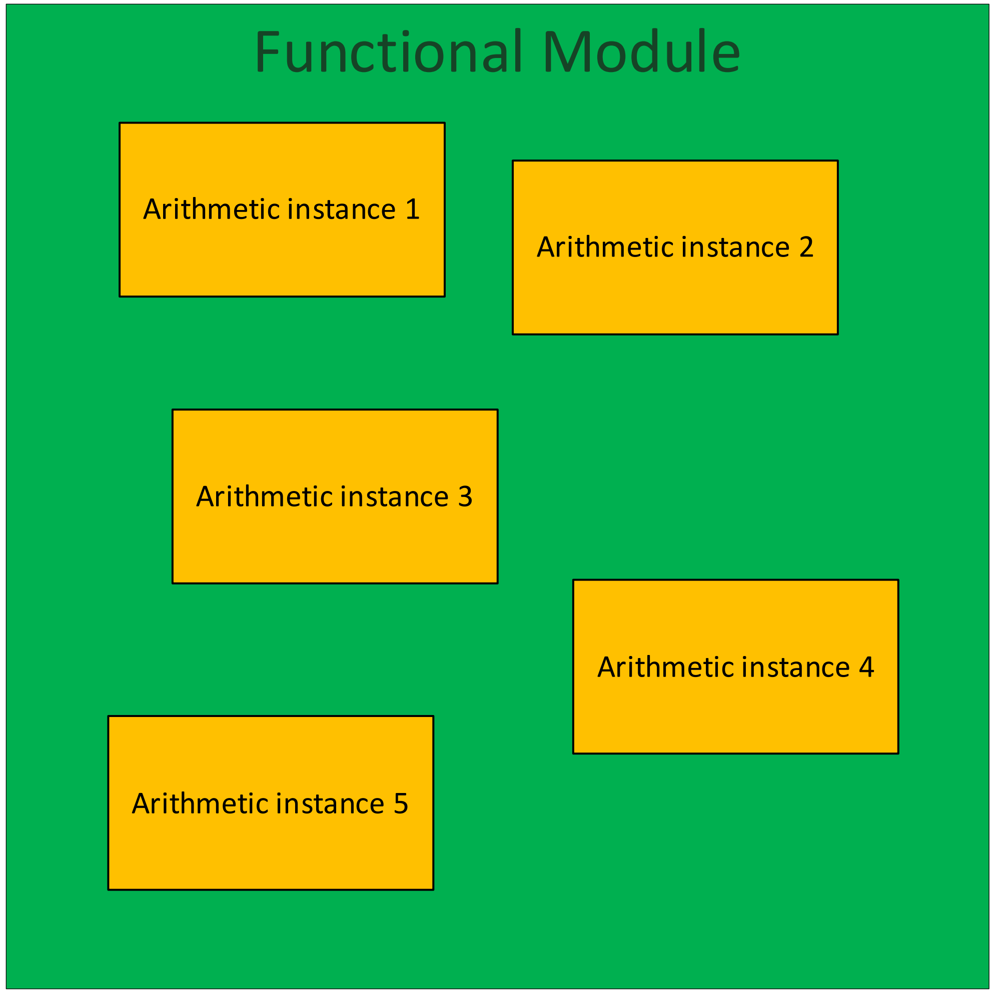
Abstracted view



Verification is done on different levels to verify the whole RPP:
- Focus on this presentation is the module verification (e.g. Func1)
- Main module verification method is constrained random verification (CRV) with UVM

Code coverage closure

Challenge on getting 100% coverage on arithmetic modules



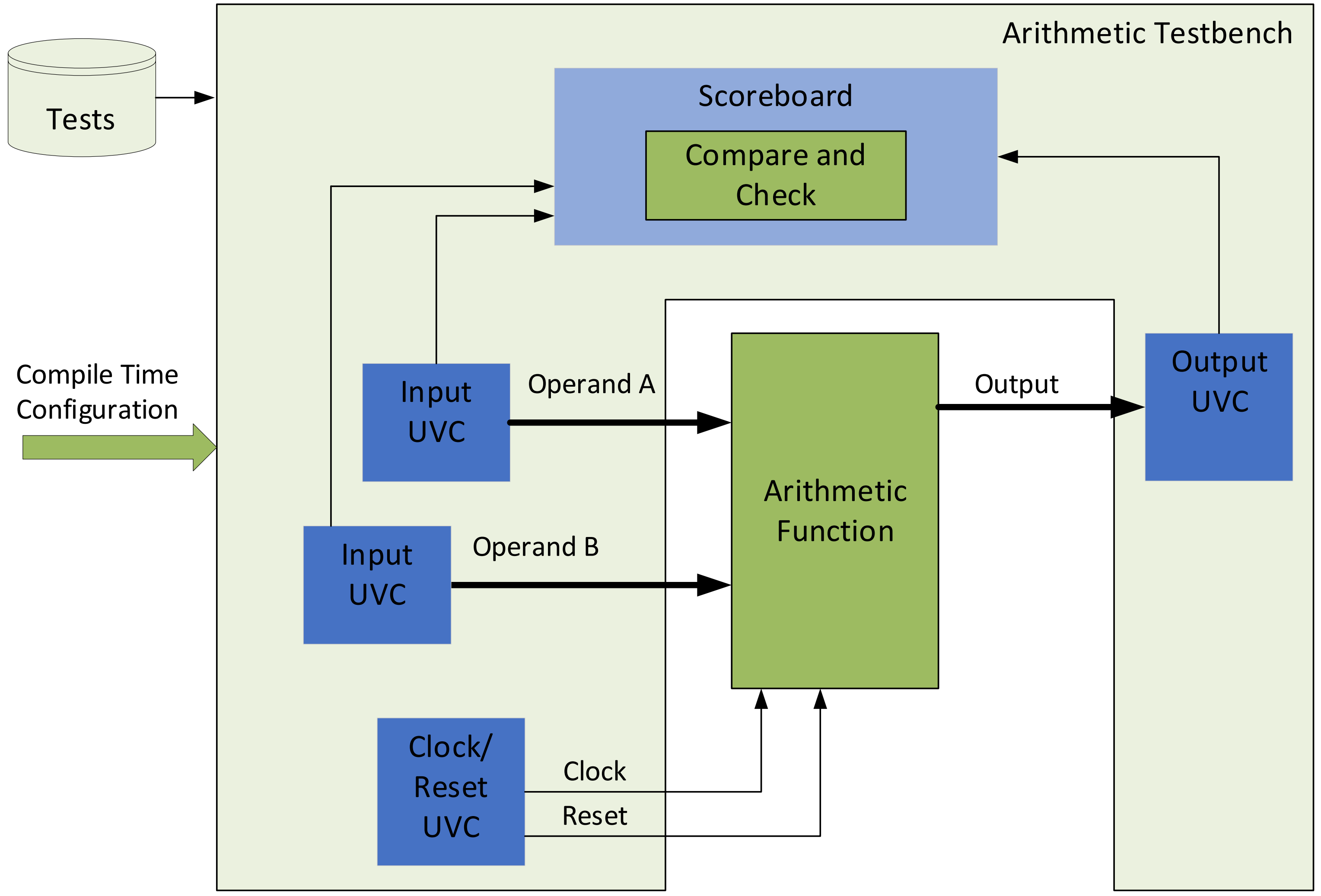
🟨	Less than 100 % code coverage
🟩	100% code coverage

Arithmetic modules:
- Math calculation like multiply, divide, add, round, etc…
- Compile time configurable (systemverilog parameters).

Code coverage holes in arithmetic modules:
- Holes that should be manually waived:
    - Limitations on the input operand values (hard to proof).
- Holes that need to be verified:
    - Too deep logic to have enough randomization.
    - Large bit widths of input operands.

Closing code coverage of arithmetic modules

Initial approach: use CRV for arithmetic modules

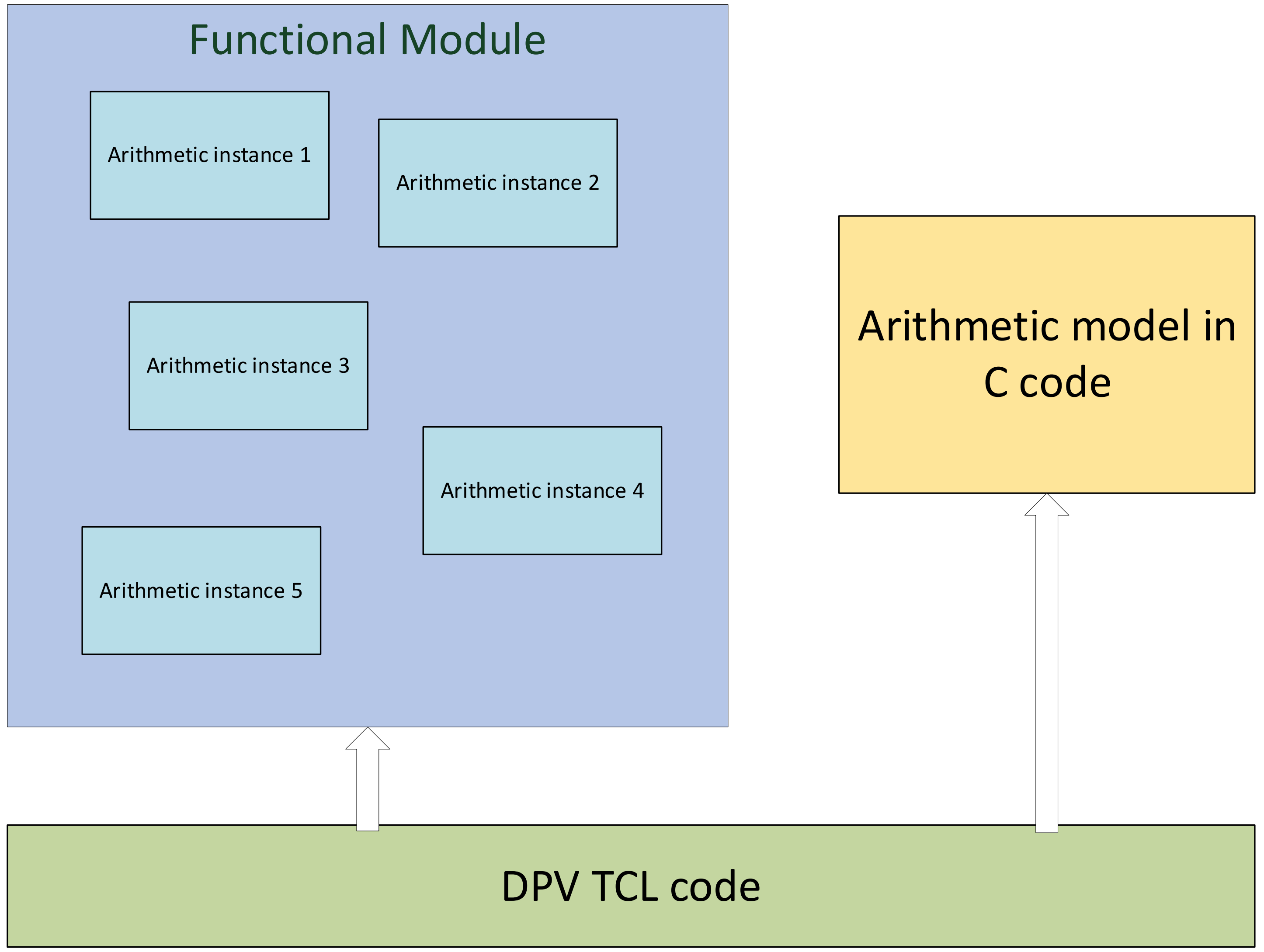


For every used arithmetic module configuration, a full CRV verification is done to reach 100% code coverage. Then it can be excluded from the module coverage metrics.

The testbench is compile time configurable, total number of configurations for RPP: 142

Disadvantages:
- Configuration parameter values are hand crafted (danger of verifying the wrong one).
- We need to blend the 142 verification reports into the RPP verification report.
- Quite some maintenance.
- Not part of the (top) module verification.

New approach: Synopsys DPV

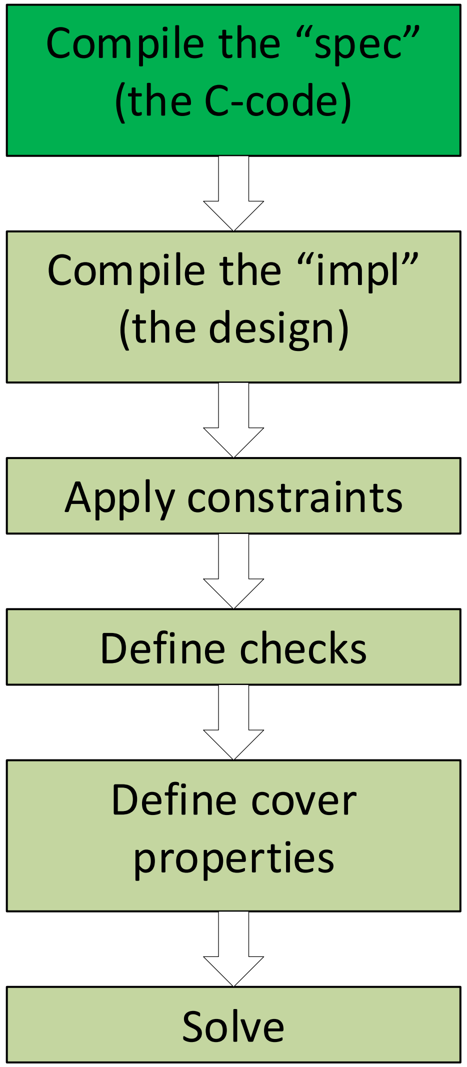


Advantages:
- Directly verify the arithmetic module(s) inside the module to be verified.
- Coverage is part of the DPV execution, no additional reporting required.
- Executed as part of the module verification.
- Compile time configuration is automatically verified (applied in the C code only).
- C-code model is re-used for other functional module testbenches.

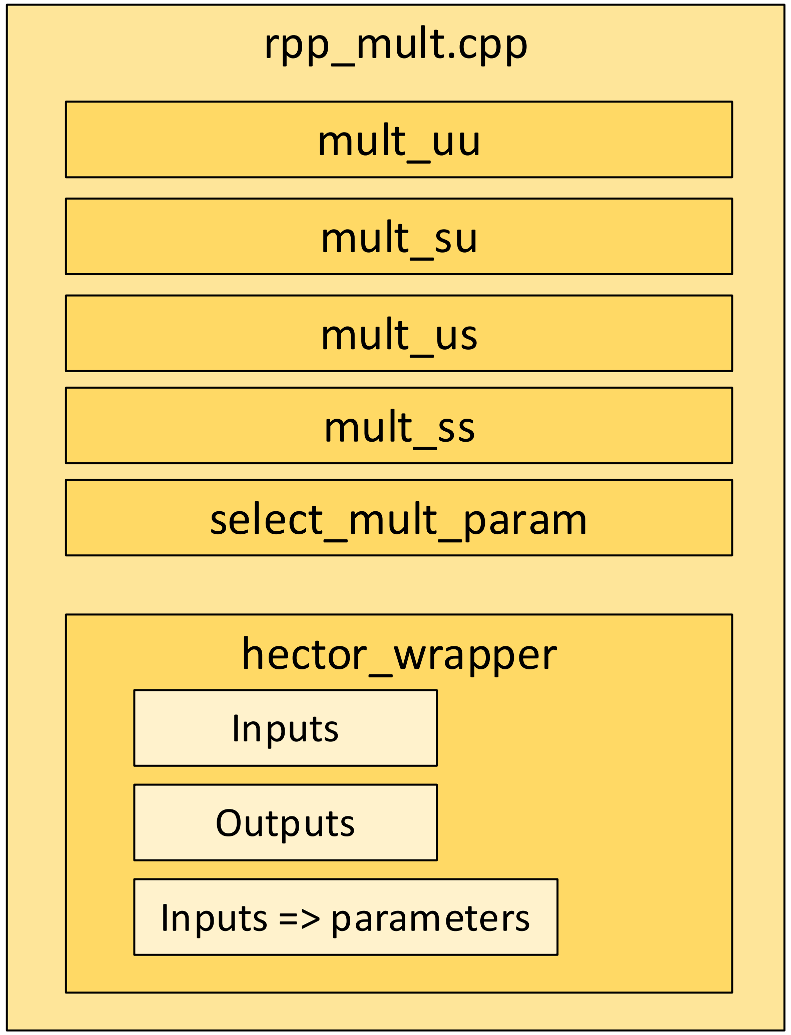
DPV flow details

DPV flow

Compilation of the C-code

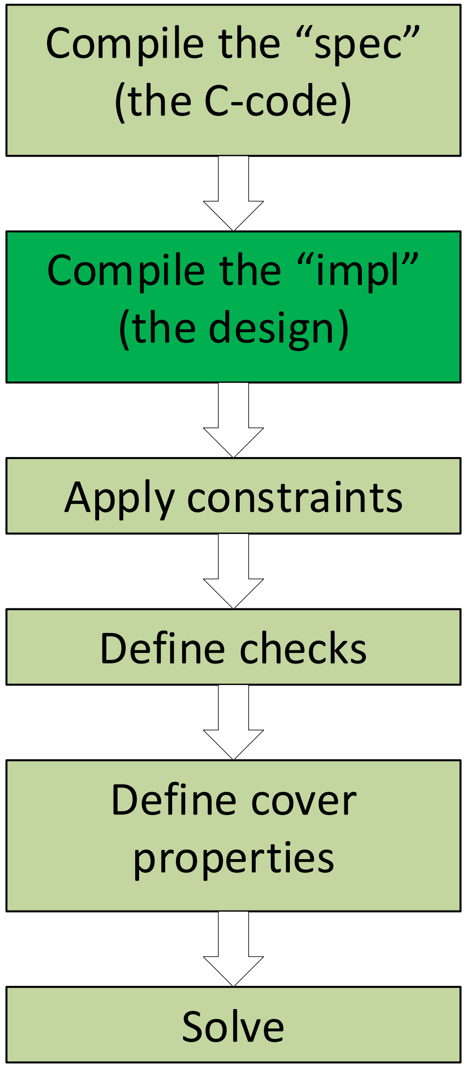

Compile the “spec” (the C-code)
Compile the “impl” (the design)
Apply constraints
Define checks
Define cover properties
Solve

Multiplier example:



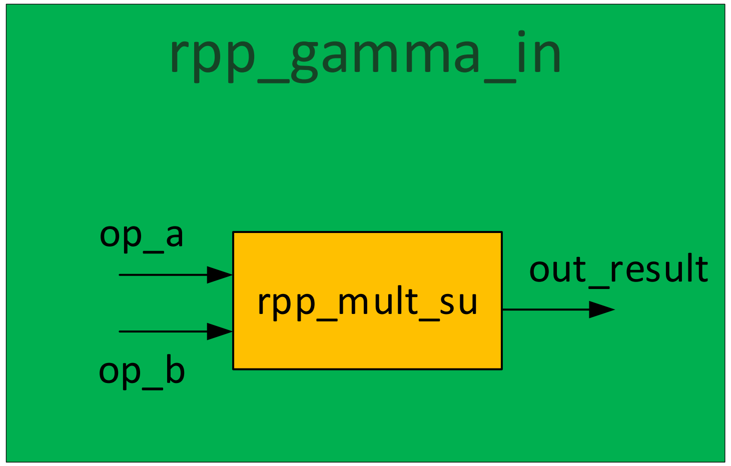
DPV flow

Compilation of the design


Compile the "spec" (the C-code)
Compile the "impl" (the design)
Apply constraints
Define checks
Define cover properties
Solve

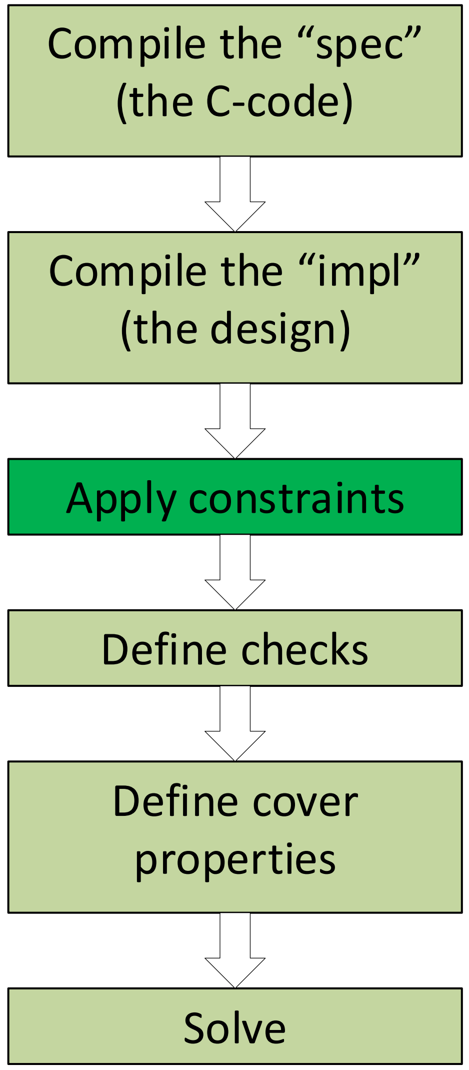
The toplevel design (module) is *rpp_gamma_in* (named *impl* in DPV)!

```
proc compile_impl {} {
  create_design -name impl -top rpp_gamma_in -clock clk -reset
reset_n -negReset -cov

  vcs -f ../../src/design.vcs.f

  compile_design impl
}
```



DPV flow

Applying constraints


Compile the “spec” (the C-code)
Compile the “impl” (the design)
Apply constraints
Define checks
Define cover properties
Solve

Constraints for the multiplier configuration in the spec:

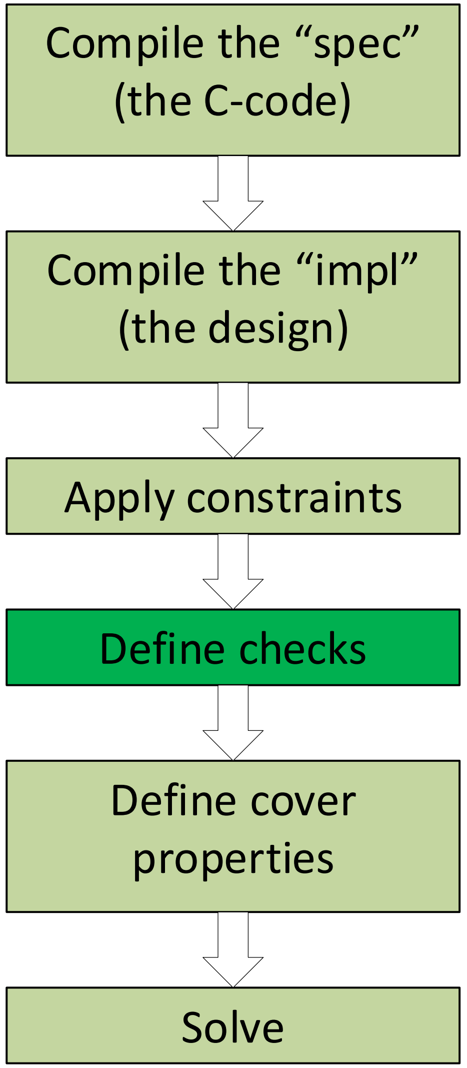
```
#align input data width (opa: 26, opb: 24)
assume spec_opa_msb_bits = -always spec.in_opa[31:26] == 6’b0
assume spec_opb_msb_bits = -always spec.in_opb[31:24] == 8'b0

#set parameters
assume spec_param_signed_opa = -always (spec.signed_opa == 1)
assume spec_param_signed_opb = -always (spec.signed_opb == 0)
assume spec_param_dw_opa = -always (spec.dw_opa == 26)
assume spec_param_dw_opb = -always (spec.dw_opb == 24)
```

Constraints for selecting the multiplier inputs:

```
#inputs
assume in_opa = -always impl.u_core…u_rpp_mult_su.in_opa == spec.in_opa
assume in_opb = -always impl.u_core…u_rpp_mult_su.in_opb == spec.in_opb
```

DPV flow

Defining checks


Compile the "spec" (the C-code)
Compile the "impl" (the design)
Apply constraints
Define checks
Define cover properties
Solve

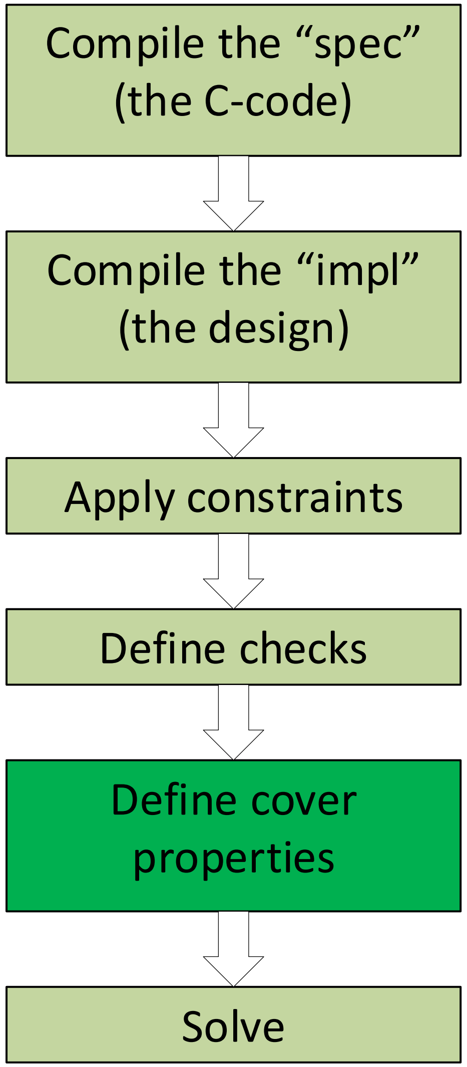
The check on the multiplier output:

```
lemma out_mult = impl.u_core...u_rpp_mult_su.out_result(3) == spec.out_result(1)
```

The multiplier design needs one clock cycle to output results.

DPV flow

Defining coverage properties

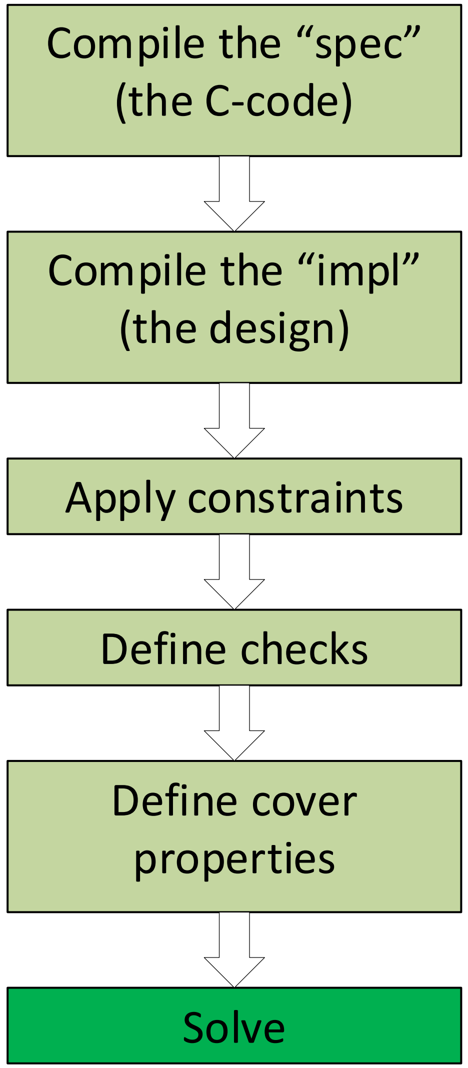

Compile the "spec" (the C-code)
Compile the "impl" (the design)
Apply constraints
Define checks
Define cover properties
Solve

Add coverage for the inputs:

```
 for {set i 0} {$i < 24} {incr i} {
     cover cover_opa_opb_zero_$i =
(impl.u_core...u_rpp_mult_su.in_opa[$i](2) == 1'b0) &&
(impl.u_core...u_rpp_mult_su.in_opb[$i](2) == 1'b0)
     cover cover_opa_opb_one_$i =
(impl.u_core...u_rpp_mult_su.in_opa[$i](2) == 1'b1) &&
(impl.u_core...u_rpp_mult_su.in_opb[$i](2) == 1'b1)
     cover cover_opa_opb_10_$i =
(impl.u_core...u_rpp_mult_su.in_opa[$i](2) == 1'b1) &&
(impl.u_core...u_rpp_mult_su.in_opb[$i](2) == 1'b0)
     cover cover_opa_opb_01_$i =
(impl.u_core...u_rpp_mult_su.in_opa[$i](2) == 1'b0) &&
(impl.u_core...u_rpp_mult_su.in_opb[$i](2) == 1'b1)
}
```

DPV flow

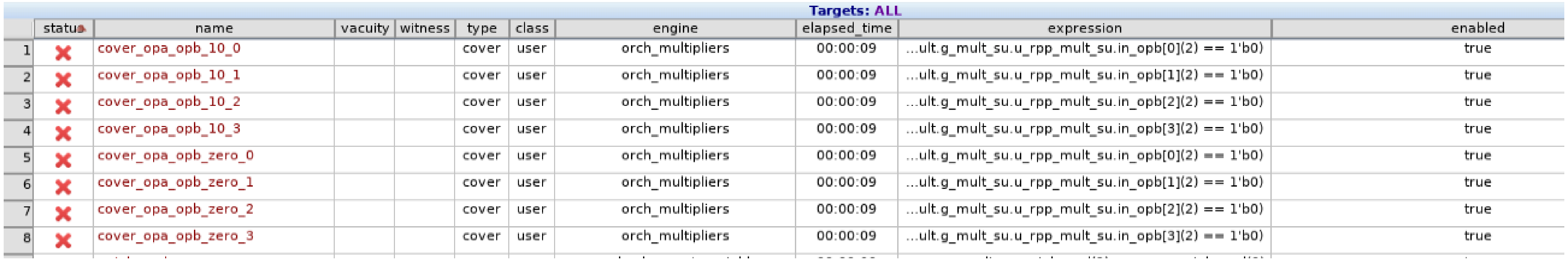
Let DPV formally proof the equivalence of the RTL vs. the C code.


Compile the “spec” (the C-code)
Compile the “impl” (the design)
Apply constraints
Define checks
Define cover properties
Solve

Example

Example: multiplier coverage

Multiplier verification that shows limitations on the inputs



Targets: ALL

	status	name	vacuity	witness	type	class	engine	elapsed_time	expression	enabled
1	✗	cover_opa_opb_10_0			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[0](2) == 1'b0)	true
2	✗	cover_opa_opb_10_1			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[1](2) == 1'b0)	true
3	✗	cover_opa_opb_10_2			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[2](2) == 1'b0)	true
4	✗	cover_opa_opb_10_3			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[3](2) == 1'b0)	true
5	✗	cover_opa_opb_zero_0			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[0](2) == 1'b0)	true
6	✗	cover_opa_opb_zero_1			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[1](2) == 1'b0)	true
7	✗	cover_opa_opb_zero_2			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[2](2) == 1'b0)	true
8	✗	cover_opa_opb_zero_3			cover	user	orch_multipliers	00:00:09	...ult.g_mult_su.u_rpp_mult_su.in_opb[3](2) == 1'b0)	true

All other coverage properties are passing:

- Operand B bits 0..3 can only be 1.

This restriction is a design limitation and should be visible in the module code coverage analysis:

- As DPV gives this proof, no further analysis required by verification.
- The design engineer should further judge if the code coverage can be waived or that he wants to update the design.

Conclusion

Conclusion

- DPV can optimize the code coverage closure of modules in the RPP.
  - Standard compile time configurable arithmetic modules can be fully verified with DPV and excluded from module code coverage analysis.
  - DPV is more efficient than using an UVM testbench to verify the arithmetic modules (TCL vs. full UVM testbench).
  - DPV implicitly verifies the correct configuration of the arithmetic module.
  - Just running the proof as a “test” in regression is enough for the verification reporting and hence needs no further effort.
  - DPV gives additional proof on limitations of the input operands of the arithmetic modules, which saves time during code coverage hole analysis.

THANK YOU

YOUR
INNOVATION
YOUR
COMMUNITY