

VC Formal Lab

Data Path Validation (DPV) App

Case-splitting and HDPS

Learning Objectives

In this lab, you will use this ALU example to learn to do the following:

- C to RTL equivalence verification methodology
- Setup tcl file to invoke VC Formal DPV
 - Setup spec and impl DUT import and configuration
 - Set up clocks and resets for RTL
 - Establish lemma to prove equivalence
- Run proof & review inconclusive result
- Apply different convergence techniques to have conclusive the proof



Lab Duration:
30 minutes

Lab Package Overview

In this lab, you will run VC Formal DPV to check a ALU with several arithmetic and logical operation to understand how to setup and run C2RTL equivalence check on DPV, and how to use the convergence techniques to have a conclusive proven result, such as case-split, assume and guarantee strategies.

Files Location

All files for this VC Formal lab are in directory:

`$VC_STATIC_HOME/doc/vcst/examples/DPV/DPV/`

Directory Structure	
DPV	Lab main directory
README_VCFormal_DPV.pdf	Lab instructions
c/	A behavioral implementation of ALU design in C/C++
rtl/	Synthesizable RTL code of the ALU design
run/	Run directory
solution/	Solution directory

Resources

The following resources are available for in-depth guidance regarding VC Formal usage, commands, and variables.

VC Formal and DPV User Guide:

`$VC_STATIC_HOME/doc/vcst/VC_Forma Docs/VC_Forma_UG.pdf`

`$VC_STATIC_HOME/doc/vcst/VC_Forma Docs/ VC_Forma_DPV_UG.pdf`

VC Formal Apps Quick References Guide:

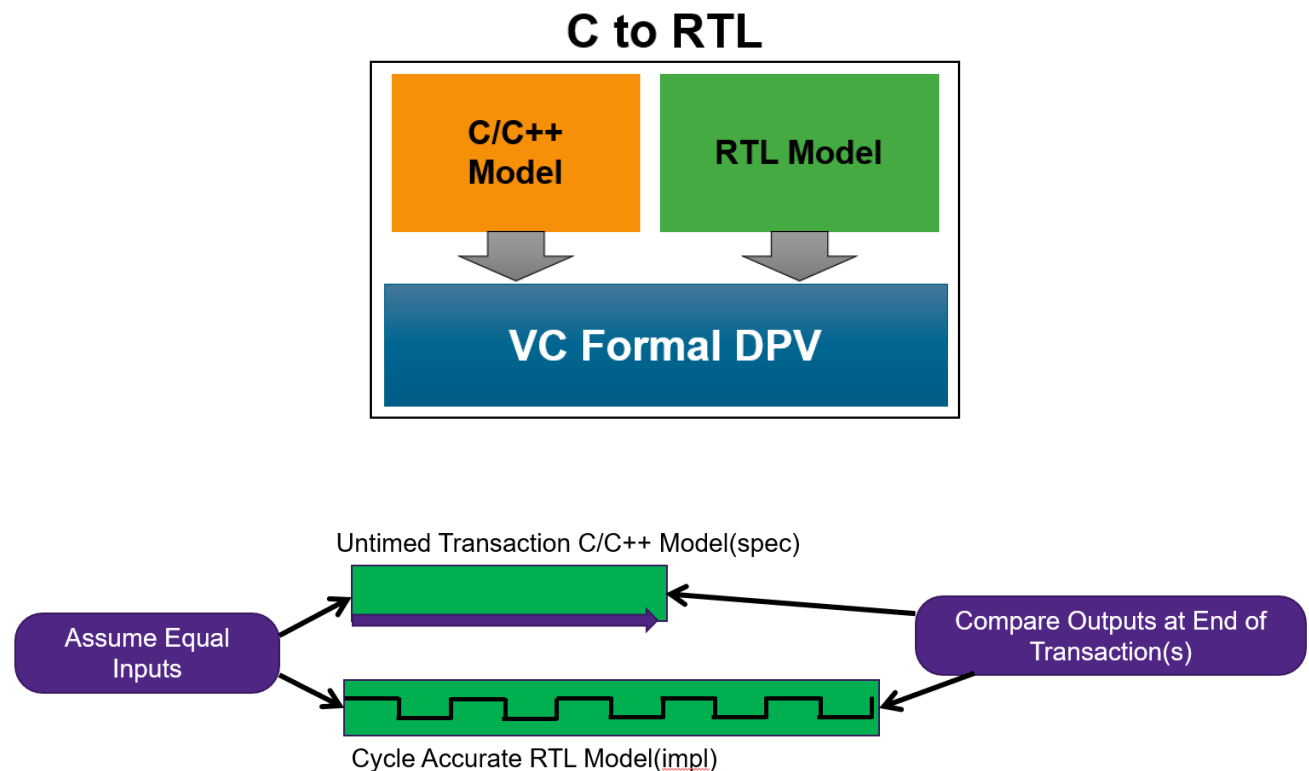
`$VC_STATIC_HOME/doc/vcst/VC_Forma Docs/VC_Forma_Quick_Reference_Guide.pdf`

VC Formal Apps Tcl Templates:

`$VC_STATIC_HOME/doc/vcst/VC_Forma Docs/Tcl_Templates/`

DUT Description

This design in the lab is to mimic currently mainstream verification methodology for C model to RTL equivalence check. The concept of this methodology is that in the first step engineers would complete a C model and verify its algorithm could meet the technical requirement and to make sure all its results are golden, then in the next step, designers would develop RTL design to match the result with golden and optimize its PPA(power, performance, area), and if there is any mismatch result, engineers need to fix the design or setup continuously till spec and impl setup have a match result. So, following this methodology, engineers could signoff the RTL design with DPV.



The DUT top is “alu”, it’s a simple ALU design with one latency from inputs to outputs, to support several arithmetic/logical operation modes with 2 inputs. It includes a C model for spec and a RTL design for impl, so users could use this Lab to practice several DPV convergence techniques to converge the proof.

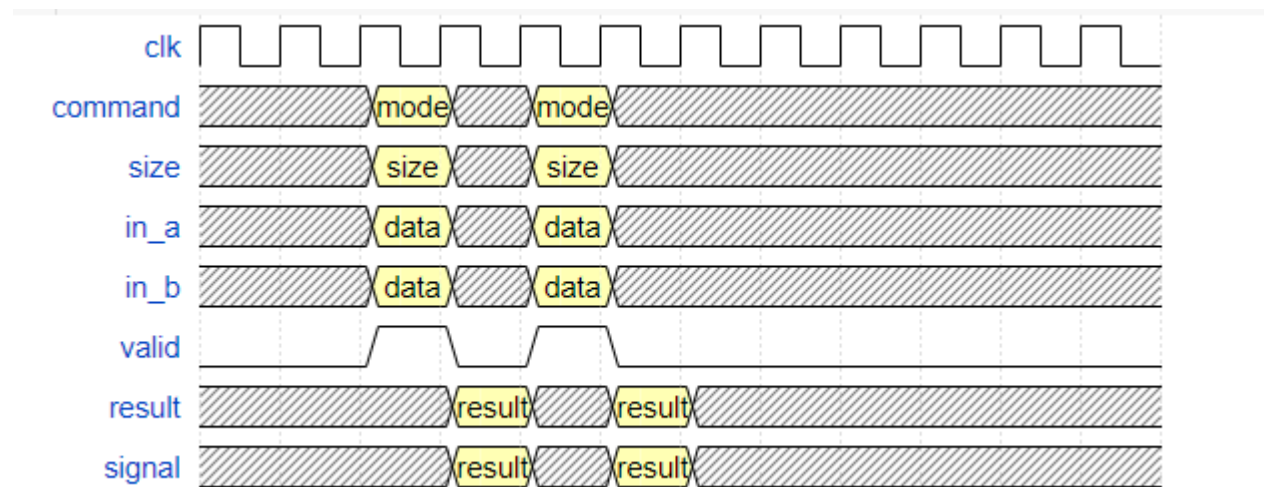
DUT in/out Port Description

Signal	Direction	Width	Description
clk	in	1	A positive edge clock
reset	in	1	A high active asynchronous reset signal
valid	in	1	A high active signal as in_a and in_b are valid
command	in	3	Command for ALU operation mode 0: AND

			1: OR 2: ADD 3: SUB 4: SATURATING ADD 5: SATURATING SUB 6: in_a*in_b 16x16 into 32 bits only Please see detail below *
size	in	1	Config operation bit width 0: 16-bit 1: 32-bit No effect as command == 6 Block is not required to clear high bits
in_a	in	32	Operand 1 of ALU input
in_b	in	32	Operand 2 of ALU input
result	out	32	Output of ALU
signal	out	2	Output flag for result status, see detail below *

- *Operation mode detail of each command:
 - 0 - AND (result all 1's sets signal[1], result all 0's set signal[0])
 - 1 - OR (result all 1's set signal[1], result all 0's sets signal[0])
 - 2 - ADD (inputs are 2's complement, overflow sets signal[1], zero result sets signal[0])
 - 3 - SUBTRACT (A-B, inputs are 2's complement, overflow sets signal[1], zero result sets signal[0])
 - 4 - SATURATING ADD (inputs are unsigned, saturation sets signal[1] and result is all 1's to proper width, zero result sets signal[0])
 - 5 - SATURATING SUBTRACT (A-B, inputs are unsigned, saturation sets signal[1] and result is all 0's, unsaturated zero result sets signal[0])
 - 6 - A*B, 16x16 into 32 bit only, zero result sets signal[0]

IMPL Data in/out Timing Diagram



Prepare your Environment

1. Load VC Formal and Verdi to set up the tool and required licenses. For example:

```
% module load vcstatic
% module load verdi
```

2. Change your working directory to run

```
% cd run
```

3. then open the file ./host.qsub using an editor, and please refer to section 6.6 in DPV_UserGuide.pdf to setup multi-processor environment

```
% vi host.qsub
```

```
#LSF example
1 | 2 | /tmp | LSF | bsub -q bnormal
#RSH example
1 | 10.10.10.10 | 4 | /tmp | RSH | rsh
```

Now you are ready to begin the lab.

Create run.tcl file for SEQ setup

VC Formal has a tcl based command interface. The most common way is to start with a tcl file to setup and compile the design. At this step, user will create a template file for the multiplier design used in this Lab.

The design files are under “c” and “rtl” directories.

4. Open the file ./run.tcl (any arbitrary name is ok to use) using an editor. e.g.

```
% vi run.tcl
```

5. Configure DPV default settings in run.tcl

```
#C++ New Front End Flow
set _hector_comp_use_new_flow true

#Enable Vacuity
set_fml_var fml_vacuity_on true

#Enable Witness
set_fml_var fml_witness_on true
```

6. Enter following commands to compile spec, it is a C model:

```
proc compile_spec {} {
    create_design -name spec -top DPV_wrapper -cov
    cppan -I. \
        ../c/alu.cpp
    compile_design spec
}
```

7. Enter following commands to compile impl design, it's a verilog RTL:

```
proc compile_impl {} {
    global step
    create_design -name impl -top alu -clock clk -reset reset
    vcs -sverilog ../rtl/multiplier_bug.sv ../rtl/alu.sv
    compile_design impl
}
```

8. Use following proc to run compile and compose both spec and impl designs. It will create a DPV formal model, DFG(data flow graphic) database for both spec and impl design so user can setup formal property such as assume and lemma on it to execute spec to impl transaction equivalence check

```
proc make {} {
    if {[compile_spec] == 0} {
        puts "Failure in compiling the specification model."
    }
    if {[compile_impl] == 0} {
        puts "Failure in compiling the implementation model."
    }
    if {[compose] == 0} {
        puts "Failure in composing the design."
    }
}
```

9. Define lemmas/assumes in the tcl procs as below, there would be 5 steps to guide users how to prove C2RTL with the techniques of convergence step by step, here is the table of objectives for each step

Step Setting	Description
1	The basic ual to prove C2RTL EQ. 1. Users need to review the falsified result and fix the design debug 2. After design fix, there are inconclusive lemmas to be solved
2	Apply case-split with “command” to prove all the other modes except for multiplication mode
3	Break down RTL multiplier into 2 smaller parts to prove them
4	Assume the previous proven with “step 2” case-plit to converge C2RTL EQ
5	Show another approach to converge the proof by applying HDPS to prove RTL multiplier result directly and then assume the proven result to prove the original C2RTL EQ

```

proc global_assumes {} {
    map_by_name -inputs -specphase 1 -implphase 1
    assume command_range = spec.command(1) <= 6
    assume multiply_size = impl.command(1) == 6 -> impl.size(1) == 0
}

proc ual {} {
    global step
    global_assumes
    if {$step == 1} {
        assume spec.command(1) <= 6
    } elseif {$step == 2} {
        assume spec.command(1) <= 6
    } elseif {$step == 3} {
        assume spec.command(1) == 6
        multiplier_properties lemma
    } elseif {$step == 4} {
        multiplier_properties assume
    } elseif {$step == 5} {
        assume mult = impl.command(1) == 6 -> impl.temp_result(1) ==
impl.in_a(1)[15:0] * impl.in_b(1)[15:0]
    }

    if {$step != 3} {
        lemma result_equal_small = impl.valid(1) && impl.size(1) == 0 ->
impl.result(3)[15:0] == spec.result(1)[15:0] || impl.command(1) == 6
        lemma result_equal_big = impl.valid(1) && (impl.size(1) == 1 ||
impl.command(1) == 6) -> impl.result(3) == spec.result(1)
        lemma signal_equal = impl.valid(1) -> impl.signal(3)[1:0] ==
spec.signal(1)[1:0]
    }
}

proc multiplier_properties {type} {
    #Add properties for multiplier. Use $type to allow properties to be
    used as assume or lemma.
    #For example, "$type name = impl.signal_a(1) == impl.signal_b(1)"
    global step
    set rel "=="
    $type mult_lo_$type = impl.mult_result_lo(1) $rel impl.in_a(1)[15:0]
* impl.in_b(1)[7:0]
    $type mult_hi_$type = impl.mult_result_hi(1) == impl.in_a(1)[15:0] *
impl.in_b(1)[15:8]
}

```


10. Define case-split strategy.

```
proc case_split {} {
    caseSplitStrategy foo
    caseEnumerate cmd -expr impl.command(1)
}
```

11. Define the proc to run all steps of proof.

```
proc run_solve {} {
    global step
    #Configure host file
    set_host_file "host.qsub"
    #Enable all multiple solve scripts
    set_hector_multiple_solve_scripts true
    set_hector_multiple_solve_scripts_list ""

    set_user_assumes_lemmas_procedure "ual"
    set_proof_name alu

    if {$step == 1} {
        #Step 1 - To prove lemma without any convergence technique
        set_hector_case_splitting_procedure ""
    }
    if {$step == 2} {
        #Step 2 - Set case split TCL procedure
        set_hector_case_splitting_procedure "case_split"
    }
    if {$step == 3} {
        #Step 3 - For command 6, To prove multiplier output
        set_hector_case_splitting_procedure ""
        set_hector_multiple_solve_scripts_list [list
orch_custom_bit_operations1]
        set_proof_name multiplier
    }
    if {$step == 4} {
        #Step 4 - Set case split TCL procedure
        set_hector_case_splitting_procedure "case_split"
    }
    if {$step == 5} {
        #Step 5 - apply HDPS multiplier result as assumption
        set_hector_case_splitting_procedure ""
    }

    gen_proof $proof_name
    check_fv
    # proofwait
}
```

```
proc run {step_in} {
    global step
    set step $step_in

    run_solve
}
```

12. Define HDPS ual and execution procs.

```
proc hdps_ual {} {
    #HDPS - Add lemma for all multiplication result, to prove it with
    HDPS
    global step
    assume impl.command(1) == 6
    assume impl.size(1) == 0
    lemma mult = impl.temp_result(1) == impl.in_a(1)[15:0] *
    impl.in_b(1)[15:0]
}

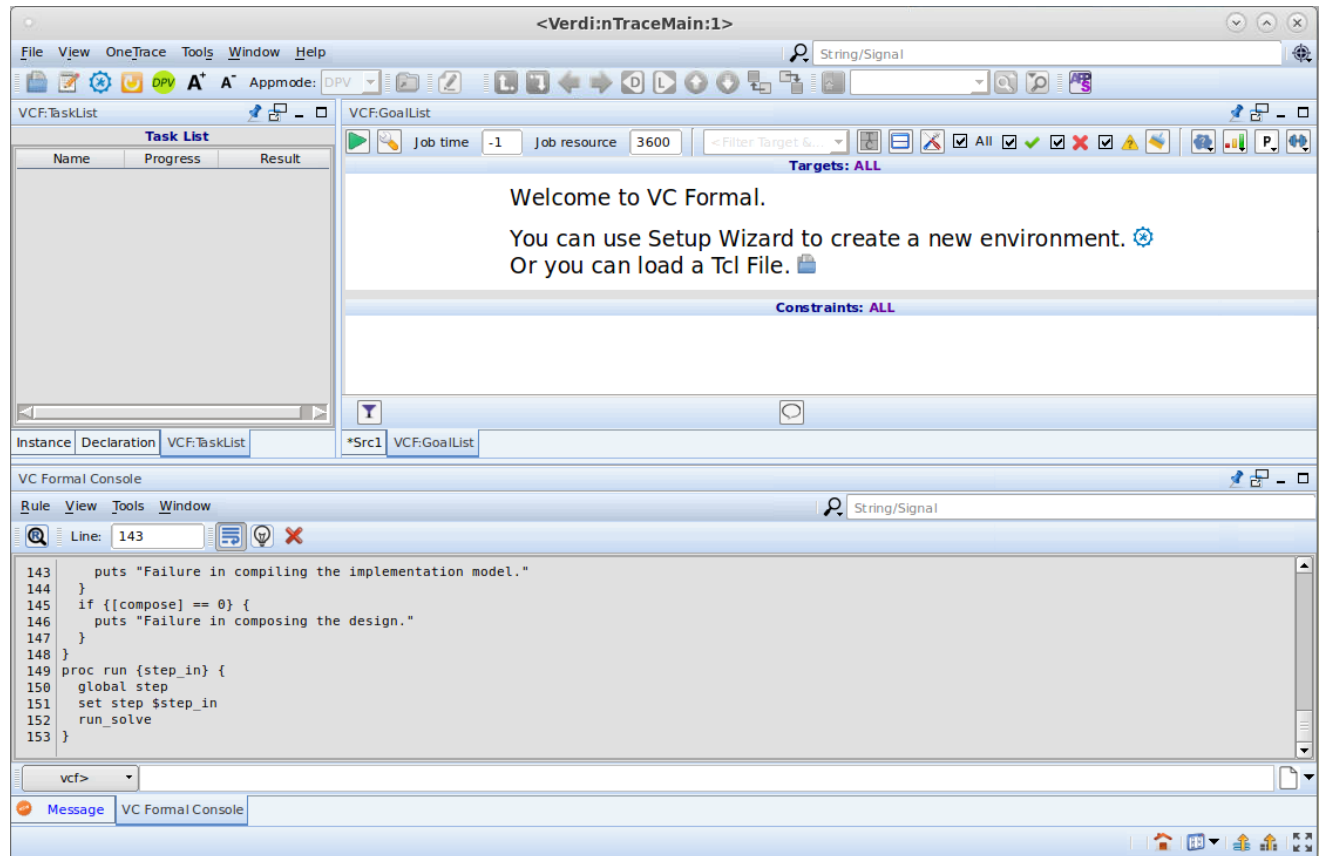
proc run_HDPS {} {
    global step
    set_hector_case_splitting_procedure ""
    set_user_assumes_lemmas_procedure "hdps_ual"
    run_all_hdps_options -encoding auto alu_hdps -modes 0 -rrtypes
    false -abstypes no_abstraction
}
```

Start VC Formal in GUI mode

13. Start the tool in GUI mode:

```
% vcf -f run.tcl -fmode DPV -gui -fml_elite
```

The GUI starts in the VC Formal mode, with icons, tables, tabs and windows especially designed for property verification.

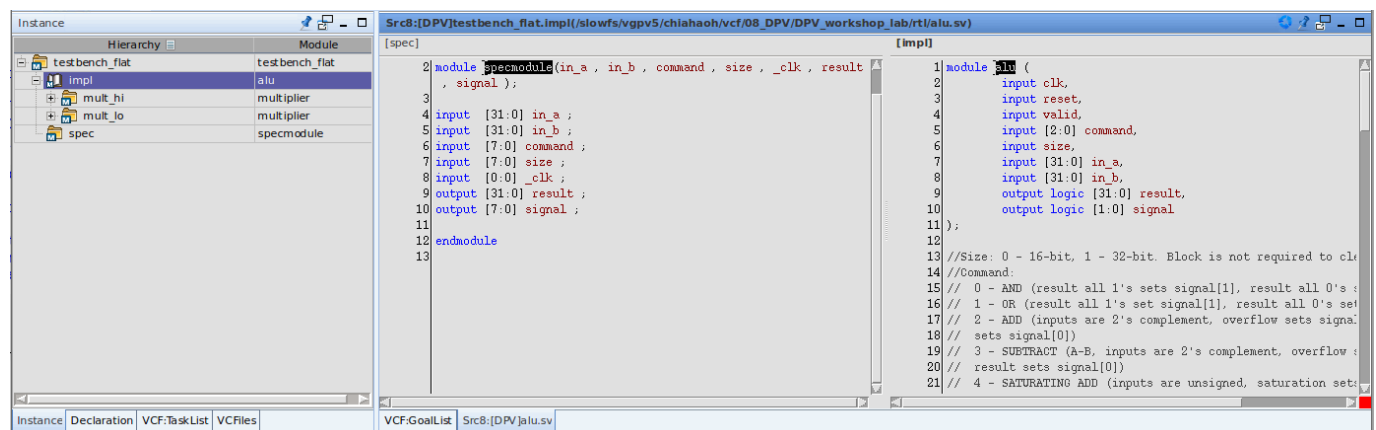


14. run make proc to compile and compose design

```
vcf > make
```

Review Design Compile Result and Information

15. Review the compiled result and make sure all DUT modules, including C and RTL, are imported completely and compiled without any error



Generate Proofs and Run for Convergence

16. Start the first step to prove the lemmas

```
vcf > run 1
```

And there are 2 lemmas have failures result as below

Name	Progress	Result
alu	3:1:2:0	Failed

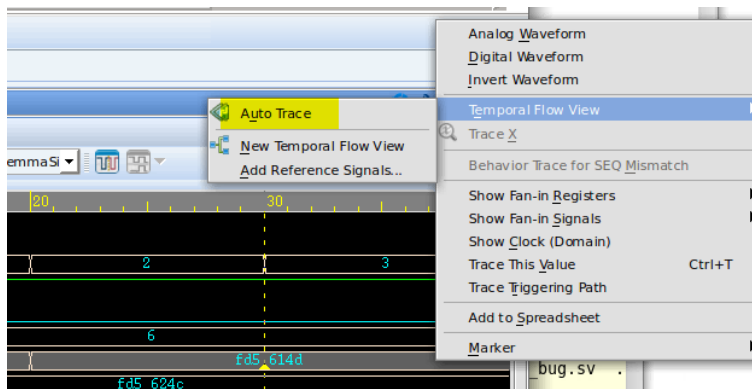
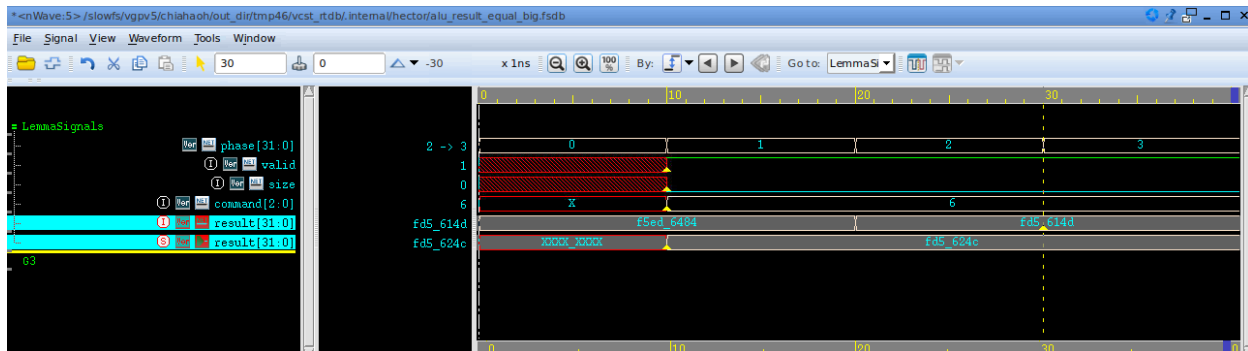
status	name	vacuity	witness	type	class	engine	elapsed_time	expression
✗	result_equal_big			lemma	user	orch_satonly	00:00:00	...mand(1) == 6 -> impl.result(3) == spec.result(1)
✓	result_equal_small			lemma	user	orch_multipliers	00:00:02	... == spec.result(1)[15:0] impl.command(1) == 6
✗	signal_equal			lemma	user	orch_multipliers	00:00:02	...ld(1) -> impl.signal(3)[1:0] == spec.signal(1)[1:0]

name	vacuity	expression
1_scv_assume_0		impl.command(1) == spec.command[2:0][1]
2_scv_assume_1		impl.in_a(1) == spec.in_a(1)
3_scv_assume_2		impl.in_b(1) == spec.in_b(1)
4_scv_assume_3		impl.size(1) == spec.size[0:0][1]

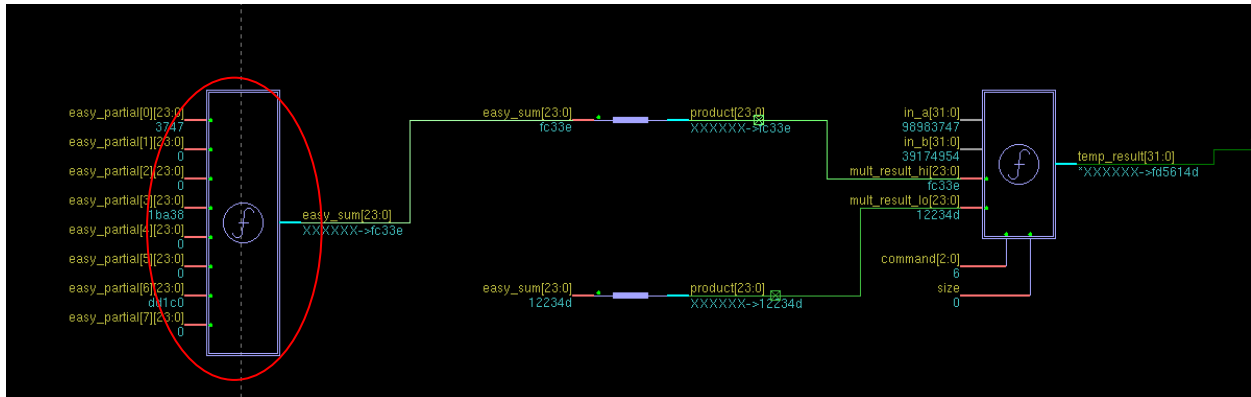
Total Properties: 3 - passed[1] - failed[2] - disabled[0]; Constraints Enabled: 7; Run Time: 0:00:08

To further debug them, double click on  to open the waveform viewer.


It shows RTL result mismatches with spec result, RMB on the value of RTL result then move to “Temporal Flow View” then select “Auto Trace”



Track it back till find the root point where unexpected result happens, then double click on it to cross probe to the corresponding RTL to understand why the mismatch happens



17. After RTL bug fixed and then rerun step 1, the lemma “result_equal_big” seems hard to prove, so here we show how to add case-split strategy for each command mode to improve convergence.

click stop icon  to stop the current run and delete “alu” proof by RMB and select “Delete Task”, then run step 2

status	name	vacuity	witness	type	class	engine	elapsed_time	expression
1	result_equal_big			lemma	user		00:00:00	...mand(1) == 6) -> impl.result(3) == sp
2	result_equal_small			lemma	user	orch_multipliers	00:00:05	... == spec.result(1)[15:0] impl.commu
3	signal_equal			lemma	user	orch_multipliers	00:00:05	...lid(1) -> impl.signal(3)[1:0] == spec.sl

name	vacuity	expression
1_scv_assume_0		impl.command(1) == spec.command[2:0](1)
2_scv_assume_1		impl.in_a(1) == spec.in_a(1)
3_scv_assume_2		impl.in_b(1) == spec.in_b(1)
4_scv_assume_3		impl.size(1) == spec.size[0:0](1)

Total Properties: 3 - passed[2] - failed[0] - disabled[0] ; Constraints Enabled: 7 ; Run Time: 0:00:08

```
vcf > run 2
```

VCF TaskList

Name	Progress	Result
alu	3:0:0:0	3:0:0:0
alu_top_(completeness)	1:1:0:0	1:1:0:0
alu_cmd_1	3:3:0:0	3:3:0:0
alu_cmd_2	3:3:0:0	3:3:0:0
alu_cmd_3	3:3:0:0	3:3:0:0
alu_cmd_4	3:3:0:0	3:3:0:0
alu_cmd_5	3:3:0:0	3:3:0:0
alu_cmd_6	3:3:0:0	3:3:0:0
alu_cmd_7	3:2:0:1	3:2:0:1
alu_cmd_8	3:0:0:0	3:0:0:0

VCF GoalList

Job time: -1 Job resource: 3600

Targets: ALL

status	name	vacuity	witness	type	class	engine	elapsed_time	expression
1	result_equal_big			lemma	user		00:00:00	...mand(1) == 6 -> impl.res
2	result_equal_small			lemma	user	orch_satonly	00:00:00	... == spec.result(1)[15:0]
3	signal_equal			lemma	user	orch_multipliers	00:00:16	...lid(1) -> impl.signal(3)[1:0]

Constraints: ALL

name	vacuity	expression
1_scv_assume_0		impl.command(1) == spec.command(2:0)[1]
2_scv_assume_1		impl.in_a(1) == spec.in_a(1)
3_scv_assume_2		impl.in_b(1) == spec.in_b(1)
4_scv_assume_3		impl.size(1) == spec.size(0:0)[1]
5_scv_assume_4		spec.command(1) <= 6

Total Properties: 3 - passed[2] - failed[0] - disabled[0]; Constraints Enabled: 8; Run Time: 0:00:46

18. After case-split for each command, for the mode “command == 6”, ALU is under multiplication mode, The lemma “result_equal_big” seems still hard to prove, so for impl RTL, break down it into the internal multiplier outputs to prove them separately, if they can be proven, then apply this result for assume and guarantee strategy to help full proof, stop the run and delete “alu” proof as previous run and run the next step

```
vcf > run 3
```

VCF TaskList

Name	Progress	Result
multiplier	2:2:0:0	2:2:0:0

VCF GoalList

Job time: -1 Job resource: 3600

Targets: ALL

status	name	vacuity	witness	type	class	engine	elapsed_time	expression
1	mult_hi_lemma			lemma	user	orch_custom_bit_operations1	00:01:23	...ult_hi(1) == impl.in_a(1)[15:0] * impl.i
2	mult_lo_lemma			lemma	user	orch_custom_bit_operations1	00:01:23	...ult_lo(1) == impl.in_a(1)[15:0] * impl.i

Constraints: ALL

name	vacuity	expression
1_scv_assume_0		impl.command(1) == spec.command(2:0)[1]
2_scv_assume_1		impl.in_a(1) == spec.in_a(1)
3_scv_assume_2		impl.in_b(1) == spec.in_b(1)
4_scv_assume_3		impl.size(1) == spec.size(0:0)[1]

Total Properties: 2 - passed[2] - failed[0] - disabled[0]; Constraints Enabled: 7; Run Time: 0:01:24

19. Since the RTL internal multipliers could be proven, then apply the proven result as assumption for the next proof, delete “multiplier” proof and run step 4

```
vcf > run 4
```

VCF TaskList

Name	Progress	Result
alu	3:0:0:0	3:0:0:0
alu_top_(completeness)	1:1:0:0	1:1:0:0
alu_cmd_1	3:3:0:0	3:3:0:0
alu_cmd_2	3:3:0:0	3:3:0:0
alu_cmd_3	3:3:0:0	3:3:0:0
alu_cmd_4	3:3:0:0	3:3:0:0
alu_cmd_5	3:3:0:0	3:3:0:0
alu_cmd_6	3:3:0:0	3:3:0:0
alu_cmd_7	3:3:0:0	3:3:0:0
alu_cmd_8	3:0:0:0	3:0:0:0

VCF GoalList

Job time: -1 Job resource: 3600

Targets: ALL

status	name	vacuity	witness	type	class	engine	elapsed_time	expression
1	result_equal_big			lemma	user	casesplit	00:00:19	...mand(1) == 6 -> impl.res
2	result_equal_small			lemma	user	casesplit	00:00:03	... == spec.result(1)[15:0]
3	signal_equal			lemma	user	casesplit	00:00:20	...lid(1) -> impl.signal(3)[1:0]

Constraints: ALL

name	vacuity	expression
1_scv_assume_0		impl.command(1) == spec.command(2:0)[1]
2_scv_assume_1		impl.in_a(1) == spec.in_a(1)
3_scv_assume_2		impl.in_b(1) == spec.in_b(1)
4_scv_assume_3		impl.size(1) == spec.size(0:0)[1]
5_command_range		spec.command(1) <= 6

Total Properties: 3 - passed[3] - failed[0] - disabled[0]; Constraints Enabled: 8; Run Time: 0:00:37

20. Now, observe that all the lemmas can be proven as above

Run with HDPS to Accelerate Prove

21. Since the bottle neck of lemma “result_equal_big” is at multiplier stage, and HDPS method could quickly prove RTL multiplication proof, so there is another run to show how it work, first, delete “alu” proof and run HDPS

```
vcf > run_HDPS
```

The screenshot shows the VCF TaskList window with a single task 'alu_hdps' in progress, showing a progress bar and the result '1:1:0:0'. The VCF GoalList window shows the 'Targets: ALL' section with a table of goals. The 'Constraints: ALL' section shows two constraints: 'impl.command(1) == 6' and 'impl.size(1) == 0'. The VCF Formal Console window shows the execution log, including the HDPS settings and the command 'Working on specified proof alu_hdps'.

status	name	vacuity	witness	type	class	engine	elapsed_time	expression
1	mult			lemma	user	orch_hdps_mode_0_nabs_auto	00:00:01	...result(1) == impl.in_a(1)[15:0]

name	vacuity	expression
1_scv_assume_0		impl.command(1) == 6
2_scv_assume_1		impl.size(1) == 0

Total Properties: 1 - passed[1] - failed[0] - disabled[0] ; Constraints Enabled: 2 ; Run Time: 0:00:02

22. The above snapshot show multiplier could be proven successfully and quickly, then change HDPS lemma to assumption for the original proofs and run step 5

```
vcf > run 5
```

The screenshot shows the VCF TaskList window with two tasks: 'alu_hdps' and 'alu'. The 'alu' task is in progress, showing a progress bar and the result '3:1:0:0'. The VCF GoalList window shows the 'Targets: ALL' section with a table of goals. The 'Constraints: ALL' section shows five constraints: 'impl.command(1) == spec.command[2:0][1]', 'impl.in_a(1) == spec.in_a(1)', 'impl.in_b(1) == spec.in_b(1)', 'impl.size(1) == spec.size[0:0][1]', and 'spec.command(1) <= 6'. The VCF Formal Console window shows the execution log, including the command 'Working on specified proof alu'.

status	name	vacuity	witness	type	class	engine	elapsed_time	expression
1	result_equal_big			lemma	user	casesplit	00:00:19	...mand(1) == 6 -> impl.res
2	result_equal_small			lemma	user	casesplit	00:00:03	... == spec.result(1)[15:0]
3	signal_equal			lemma	user	casesplit	00:00:20	...lid(1) -> impl.signal(3)[1:0]

name	vacuity	expression
1_scv_assume_0		impl.command(1) == spec.command[2:0][1]
2_scv_assume_1		impl.in_a(1) == spec.in_a(1)
3_scv_assume_2		impl.in_b(1) == spec.in_b(1)
4_scv_assume_3		impl.size(1) == spec.size[0:0][1]
5_command_range		spec.command(1) <= 6

Total Properties: 3 - passed[3] - failed[0] - disabled[0] ; Constraints Enabled: 7 ; Run Time: 0:00:04

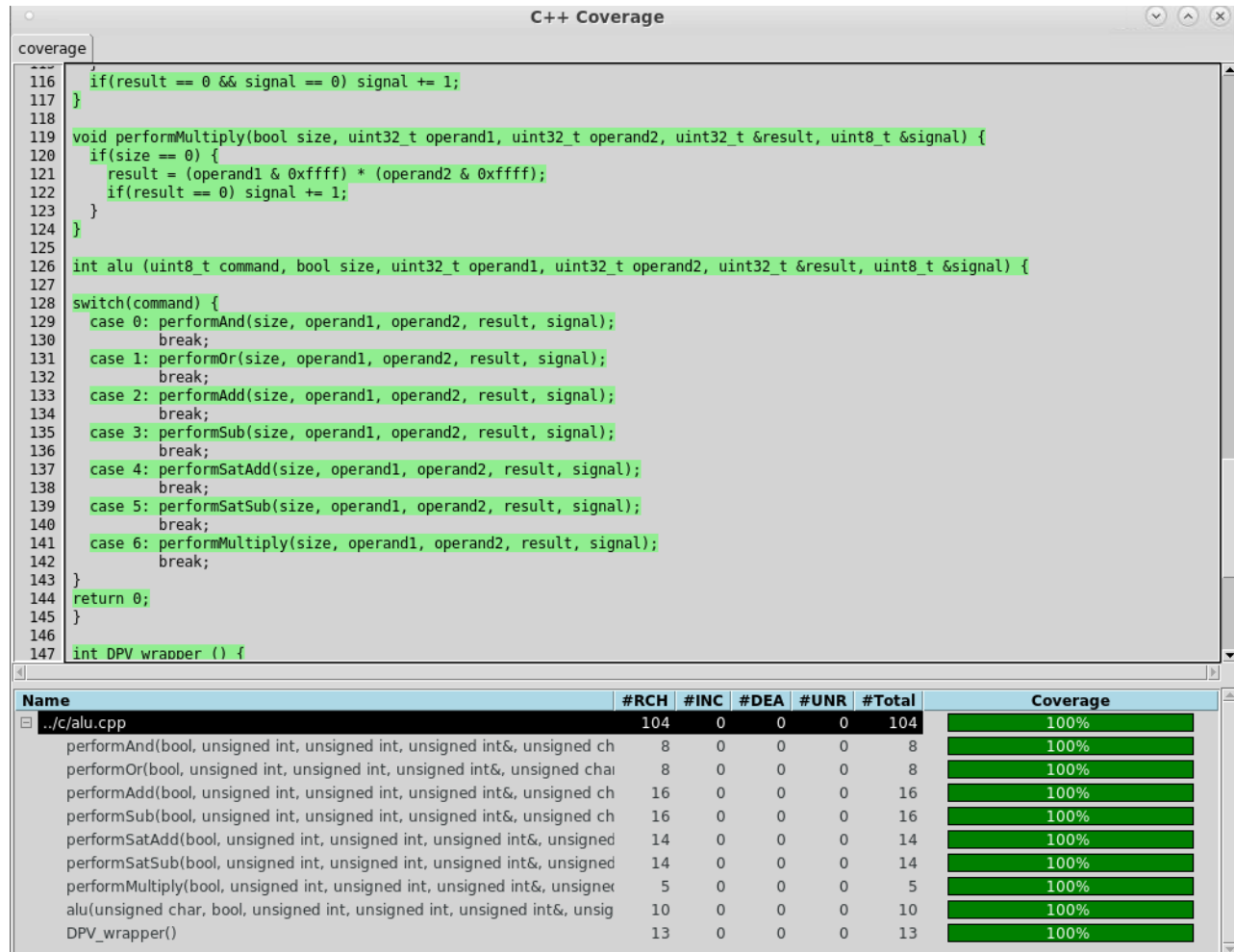
23. Observe that the original lemmas are now proven quickly

Show C Model Coverage Report

24. After all the lemmas proven, user can further review the C code coverage by this command

```
vcf > check_cpp_coverage
```

As the “C++ Coverage” window pops up, click on “alu.cpp” to review the line coverage as below to observe every function has 100% coverage



The screenshot shows the "C++ Coverage" window with the "coverage" tab selected. The code for `alu.cpp` is displayed, showing functions like `performMultiply`, `performAnd`, `performOr`, `performAdd`, `performSub`, `performSatAdd`, `performSatSub`, and `performMultiply`. Below the code, a table summarizes the coverage for each function.

Name	#RCH	#INC	#DEA	#UNR	#Total	Coverage
./c/alu.cpp	104	0	0	0	104	100%
performAnd(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	8	0	0	0	8	100%
performOr(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	8	0	0	0	8	100%
performAdd(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	16	0	0	0	16	100%
performSub(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	16	0	0	0	16	100%
performSatAdd(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	14	0	0	0	14	100%
performSatSub(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	14	0	0	0	14	100%
performMultiply(bool, unsigned int, unsigned int, unsigned int&, unsigned char)	5	0	0	0	5	100%
alu(unsigned char, bool, unsigned int, unsigned int, unsigned int&, unsigned char)	10	0	0	0	10	100%
DPV_wrapper()	13	0	0	0	13	100%