# VC Formal Lab
# Data Path Validation (DPV) App
# Setup and Standard Usage

## Learning Objectives

In this lab, you will use the DPV to verify RTL which implements floating-point fused multiply-add (in 16, 32 or 64 bit formats). In order to explain this process in some detail, we have provided a sample RTL design which implements the multiply-add operation and an actual DPV command script which was used to verify this design. To verify your RTL, you require the following things:

- Prepare your Formal environment
- Create tcl command script for DPV setup
- Start VC Formal GUI in DPV-mode
- Review Design information and Setup
- Review Setup and Generate proofs
- Run DPV formal proofs and Review Results
- Debug Failures
- Correct Errors
- Restart the Run and Verify the fix

Familiarity of basic formal verification concept is required for this lab.

**Lab Duration:**
**30 minutes**

## Files Location

All files for this VC Formal lab are in directory:
$VC_STATIC_HOME/doc/vcst/examples/DPV/

| Directory Structure | |
|---|---|
| $VC_STATIC_HOME/doc/vcst/examples/DPV | Current working directory |
| README_VCFormal_DPV.pdf | Lab instructions |
| c/ | A behavioral implementation of floating point fused multiply-add in C/C++. |
| softfloat-3e/ | Publicly available reference design in the Berkeley SoftFloat library: http://www.jhauser.us/arithmetic/SoftFloat.html |
| rtl/ | Synthesizable RTL code of the floating-point fused multiply-add (in 16, 32- or 64-bit formats) |
| run/ | Run directory |
| tcl/ | Solution of command scripts in 16, 32- or 64-bit formats |
| DPV_FMA_tutorial.docx | Details how to use DPV to verify RTL which implements floating-point fused multiply-add (in 16, 32- or 64-bit formats) |
| Design.docx | Describes the design of the floating-point multiply-add RTL and C model. |

## 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_Formal_Docs/VC_Formal_UG.pdf
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/DPV_UserGuide.pdf

VC Formal Apps Quick References Guides:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/

VC Formal Apps Tcl Templates:
$VC_STATIC_HOME/doc/vcst/VC_Formal_Docs/Quick_Reference_Guides/vcf_tcl_templates/

## Prepare your Formal Environment

1. Ensure VC Formal and Verdi is set up in your terminal, and with required licenses.

   VC Formal uses the pivotal environment variable: VC_STATIC_HOME. This variable must be set to point to the installation directory as shown in the following code snippet. In the installation directory, you can find the bin, lib, doc and other directories.

   ```
   % setenv VC_STATIC_HOME /tools/synopsys/vcst
   ```

   You can add $VC_STATIC_HOME/bin to your $PATH. To start the VC Formal tool, execute the following command:

   ```
   % $VC_STATIC_HOME/bin/vcf
   ```

   This command starts a VC Formal shell session and you see the following prompt:

   %vcf>

   The %vcf shell calls the vc_static_shell shell internally. The %vcf shell supports all the options that the vc_static_shell supports. The %vcf shell automatically runs in the 64-bit mode, unless you explicitly specify the -mode32 option.

2. Change your working directory to run.

   ```
   $> cd run
   ```

Now you are ready to begin the lab.

## Create tcl command script for DPV 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 command_script_mul*.tcl.

Where * can be add16, add32, add64 used for multiply-add unit and * can be 16, 32, 64 floating point multiplication unit verification.

The design files and filelist are located under rtl.

1. Open a new file command_script_muadd16.tcl (consider fused FP 16bit) using an editor. For example,

   ```
   % vi command_script_muadd16.tcl
   ```

2. To enable the C++11 front-end, the following command must be placed in the DPV setup file.

   ```
   set _DPV_comp_use_new_flow true
   ```

## Prepare your Formal Environment

3. Compiling RTL design. Please refer to section 4.4

```
proc compile_impl {} {
    create_design -name impl -top muladd -clock clock -
reset resetN -negReset


    set_cutpoint muladd.mpier_mantissa_0a
    set_cutpoint muladd.mpcand_mantissa_0a


    vcs -sverilog -pvalue+SIZE=16 -f
../rtl/files_muladda
```

```
    compile_design impl
}
```

4. Compiling a C/C++ Design. Please refer to section 4.3.

```
set _DPV_softfloat_version custom
proc compile_spec {} {
    create_design -name spec -top DPV_wrapper
    cppan -I../softfloat-3e/source/include \
        -I../softfloat-3e/source/8086 \
   -I../softfloat-3e/build/DPV \
        ../c/madd16.cc \
        ../softfloat-3e/source/f16_mulAdd.c \
        ../softfloat-3e/source/f32_mulAdd.c \
        ../softfloat-3e/source/f64_mulAdd.c \
        ../softfloat-3e/source/s_mulAddF16.c \
        ../softfloat-3e/source/s_mulAddF32.c \
        ../softfloat-3e/source/s_normSubnormalF16Sig.c
\
        ../softfloat-3e/source/s_normSubnormalF32Sig.c
\
        ../softfloat-3e/source/s_normSubnormalF64Sig.c
\
        ../softfloat-3e/source/s_shortShiftRightJam64.c
\
        ../softfloat-3e/source/s_countLeadingZeros32.c
\
        ../softfloat-3e/source/s_roundPackToF16.c \
        ../softfloat-3e/source/s_roundPackToF32.c \
        ../softfloat-3e/source/s_roundPackToF64.c \
        ../softfloat-3e/source/s_shiftRightJam32.c \
        ../softfloat-3e/source/s_shiftRightJam64.c \
        ../softfloat-3e/source/ARM-
VFPv2/s_propagateNaNF16UI.c \
        ../softfloat-3e/source/ARM-
VFPv2/s_propagateNaNF32UI.c \
        ../softfloat-3e/source/ARM-
VFPv2/s_propagateNaNF64UI.c \
```

```
        ../softfloat-3e/source/ARM-
VFPv2/softfloat_raiseFlags.c \
        ../softfloat-3e/source/s_countLeadingZeros8.c \
        ../softfloat-3e/source/s_countLeadingZeros16.c
\
        ../softfloat-3e/source/s_countLeadingZeros64.c
\
        ../softfloat-3e/source/s_mul64To128M.c \
        ../softfloat-3e/source/softfloat_state.c
    compile_design spec
}
```

5. Define lemmas/assumes in a tcl proc. Please refer to section 2.5.

```
proc ual {} {
    assume impl.go(1) == 1

    map_by_name -inputs -specphase 1 -implphase 1

    assume spec.rounding_mode(1) < 4

    assume impl.product_mantissa_0(3) ==
impl.mpier_mantissa_0a(3) * impl.mpcand_mantissa_0a(3)

    set_resource_limit 36000
    set_DPV_multiple_solve_scripts true
    set_DPV_multiple_solve_scripts_list [list
orch_multipliers]

    lemma rslt = spec.result(1) == impl.result(7)
    lemma ex = spec.exceptions(1) == impl.exceptions(7)
    }



proc hdps_ual {} {
    cutpoint mpier = impl.mpier_mantissa_0a(1)
    cutpoint mpcand = impl.mpcand_mantissa_0a(1)
```

```
    lemma check_mul = impl.product_mantissa_0(1) ==
mpier * mpcand
    lemma check_mul_fail = impl.product_mantissa_0(1)
!= mpier * mpcand
}
```

6. Define casesplit strategies. Please refer section 8.3.

```
proc case_split_16 {} {
    caseSplitStrategy basic

    caseBegin dnorm_norm_16
    caseAssume (spec.multiplier(1)[14:10] == 5'h00)
    caseAssume (spec.multiplicand(1)[14:10] != 5'h00)
    caseAssume (spec.multiplicand(1)[14:10] != 5'h1f)

    caseBegin norm_dnorm_16
    caseAssume (spec.multiplier(1)[14:10] != 5'h00)
    caseAssume (spec.multiplier(1)[14:10] != 5'h1f)
    caseAssume (spec.multiplicand(1)[14:10] == 5'h00)

    caseBegin A_inf_NaN_16
    caseAssume (spec.multiplier(1)[14:10] == 5'h1f)

    caseBegin B_inf_NaN_16
    caseAssume (spec.multiplicand(1)[14:10] == 5'h1f)

    caseBegin dnorm_dnorm_16
    caseAssume (spec.multiplier(1)[14:10] == 5'h00)
    caseAssume (spec.multiplicand(1)[14:10] == 5'h00)

    caseBegin norm_norm_16
    caseAssume (spec.multiplier(1)[14:10] != 5'h00)
    caseAssume (spec.multiplier(1)[14:10] != 5'h1f)
    caseAssume (spec.multiplicand(1)[14:10] != 5'h00)
    caseAssume (spec.multiplicand(1)[14:10] != 5'h1f)
```

```
}
```

## Start VC Formal GUI in DPV-mode

7. Start VC Formal GUI in DPV mode:

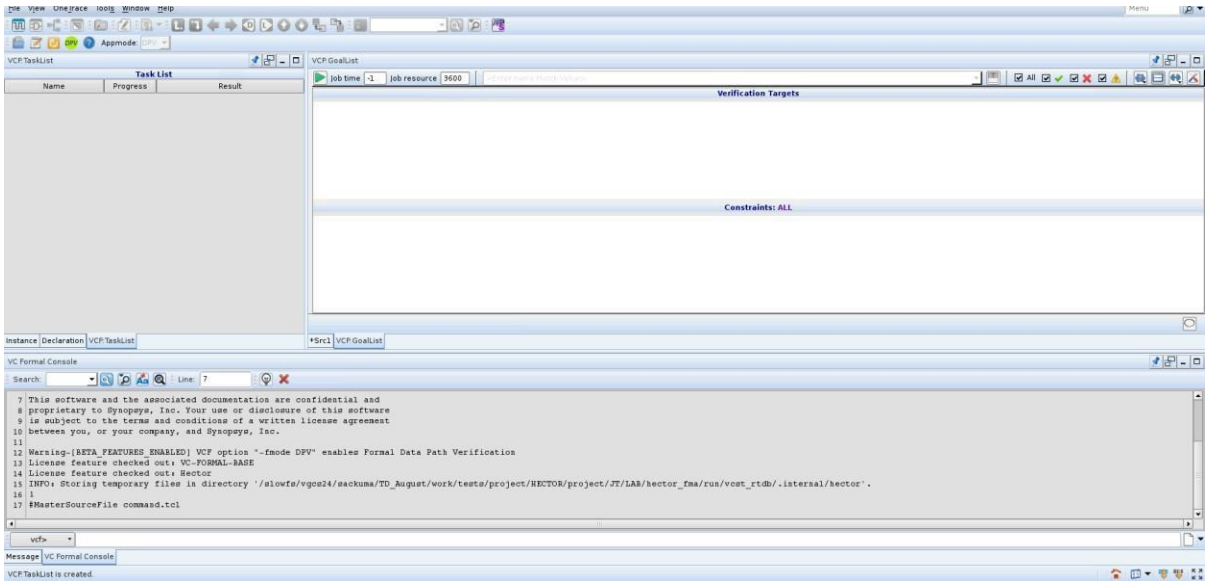
```
%vcf -verdi -f command_script_muadd16.tcl -fmode DPV &
```

DPV uses DPV technology, so currenlty we can only invoke DPV via shell command prompt. Initial configuration is shown with **Task List** on the top left, VCF Goal List table on the top right, and the VCF Shell at the bottom. Source files and proofs/lemmas are populated after compose, gen_proof/solveNB commands respectively.

## Review Design Information and Setup

8. Review design information and setup: 

on top menu bar ensuring you are in DPV mode, **Task List** on the left (you should see proofs p, hdps), **Verification Targets** on right side (you should see lemmas, assumptions) and VCF Shell at the bottom (you should see message "Finished) to ensure that setup and file parsing, elaboration, compose is successful.




## Review Setup and Generate proofs

9. Since we missed calling defined procs and proof generation it results in empty task and goal list. Click on the Edit Tcl Project File icon on the upper left. Click on Edit to activate editing. Please add below statements in the tcl command script.

```
proc make {} {
```

```
    compile_spec
    compile_impl
    compose
}

#set_host_file "host.qsub"   //Please refer to 6.6
DPV Multi Processor Env..

proc run_hdps {} {
set_user_assumes_lemmas_procedure "hdps_ual"

set_custom_solve_script "orch_custom_bit_operations"
set_DPV_rew_use_dps_engine true
set_DPV_rew_dps_solve_script
__DPV_orch_custom_dps2
set_resource_limit 200
set_DPV_rew_dps_resource_limit 1200

    run_all_hdps_options -encoding [list radix4booth]
hdps -modes 0 -rrtypes false -abstypes no_abstraction
    proofwait
}

proc run_main {} { set_user_assumes_lemmas_procedure "ual"
    set_DPV_case_splitting_procedure "case_split_16"
    solveNB p
    proofwait
}
```

10. Save the edited command_script_muadd16.tcl file: Click 

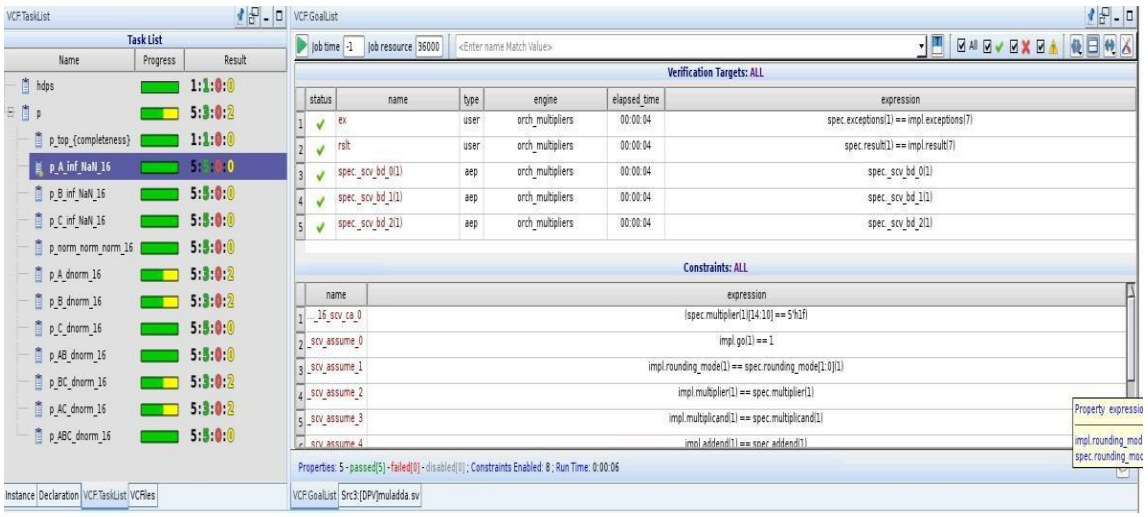
.
11. Restart VCST: click on  to restart.

## Run DPV Formal Proofs and Review Results

12. Now that you have a proper DPV setup, Execute below commands step by step:

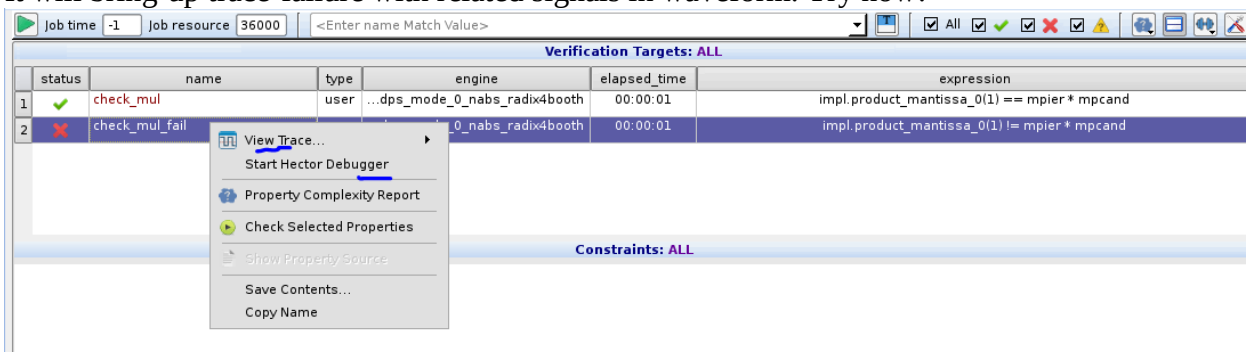
```
% make
% run_hdps
% run_main
```

Observe: **Task List** now shows 2 proofs “hdps” and “p”. Proof “p” has subproofs due to case split. scroll down on **Verification Targets** and it displays one lemma in “hdps” in proven state and if you double click proof “p” become active and you can check lemmas and its status on the **Verification Targets** tab.




## Debug Failures

13. To Debug failure in GUI: Right click on ✖ (falfisied lemma) and select “View Trace”. It will bring-up trace-failure with related signals in waveform. Try now!




14. Alternatively, C/C++ code can be debugged using ddd interface,

```
% simcex -gdb1 <failed lemma name>
```

Please refer section 7 for debugging Failed Lemmas.

## Restart the Run and Verify Fix

15. Restart VCST: click on to restart.
16. Observe in Verification Targets there are no falsified properties, and all are proven.