NVDLA Verification Suite User Guide

Contents

Preparation

Acronym

Following acronyms will be frequently used in this document, if there are abbreviations you don't know their meanings, please check this chapter.

Acronym	Description
tree	Repository hosting NVDLA source code
TOT	Top of Tree, refer to the root of NVDLA HW repository nvdla/hw
OUTDIR	Generated code under TOT/ourdir
Sub-unit	A module which has a dedicated register block. For example, CDMA/CSC/CMAC/CACC/SDP/SDP_RDMA are all sub-units.
Resource	Hardware operation resources. One resource may contain one or more sub-units. For example, a combination of CSC/CMAC/CACC is one resource. SDP_RDMA and SDP are two independent resources.
Hardware pipeline	A combination of resources for a specified operation. A hardware pipeline starts with one or more read DMA(s) and ends with one write DMA.
Scenario	A description of hardware pipelines.

Acronym	Description
HWL/HW layer	One complete hardware processing within a hardware pipeline. A hardware layer starts with a set of register configuration with an enable field. When it's done, it triggers ONE interrupt. A hardware layer requires a hardware pipeline.
Image data	The input data of convolution HW layer. In convolution, image data is the station operand.
Feature data	The input/output data of HW-layer. Feature data is generated from one HW-layer and can be used as input data for next HW-layer. In convolution, feature data is the station operand.
Weight data	The moving operand of convolution operation.

Setup tree.make and Build tree

Please follow instructions in NVDLA Environment Setup Guide (environment_setup_guide.html), section *Tools and Dependency Libraries Setup* to setup environment and section *Build Test Bench* to build test bench before advancing to next section.

Quick Start

After tree was built, we can run a test and a regression to validate NVDLA environment is in good health.

Let's create a directory named health_exam.

```
TOT> mkdir health_exam

TOT> cd health_exam
```

Running single test

Let run a convolution tests

```
TOT/health_exam> TOT/verif/tools/run_test.py -P nv_small dc_24x33x55_5x5x55x25_int8_0 -outdir
dc_24x33x55_5x5x55x25_int8_output -wave -v nvdla_utb
```

Argument explanations:

Argument	Description
-P nv_small	Running test on project NV_SMALL
dc_24x33x55_5x5x55x25_int8_0	Select test source named dc_24x33x55_5x5x55x25_int8_0
-outdir dc_24x33x55_5x5x55x25_int8_output	Simulation will be run in directory dc_24x33x55_5x5x55x25_int8_output will be
-wave	Enable waveform dumping
-v nvdla_utb	Specify target testbench

Wait for a while, when simulation is finished, following message will be shown in the end of log:

```
******************************
**        TEST PASS           **
******************************


PPPP     A     SSSSS  SSSSS
P   P   A A    S      S
PPPP   AAAAA   SSSSS  SSSSS
P     A     A      S      S
P     A     A  SSSSS  SSSSS
```

There are several files were generated under

```
TOT/health_exam/dc_24x33x55_5x5x55x25_int8_output
```

1. run_verdi.sh: which is used to kick-off Verdi to view simulation waveform
2. testout: which contains output logs

Running a list of tests

We can run a list of tests for wider health status check

```
TOT/health_examp>TOT/verif/tools/run_plan.py -P nv_small -tp nv_small -atag protection -no_lsf -run_dir
protection_tests -monitor
```

Argument explanations:

Argument	Description
-P nv_small	Running test on project NV_SMALL
-tp nv_small	Running tests in test plan nv_small
-atag protection	Select tests which have been tagged with “protection”
-no_lsf	Use local CPU to run tests
-run_dir protection_tests	Output file will be run in directory protection_tests
-monitor	Continuously monitoring test running status, until all simulations are done or reach maximum runtime

**detail meanings of arguments could be found in both script self-contained help argument and later section* Regression Tool Set

Terminal will update tests status in a certain interval:

```
[INFO] Dir = TOT/health_exam/protection_tests
-------------------------------------------------------------------------------------------------
Test                                TB                  Status     Errinfo
-------------------------------------------------------------------------------------------------
pdp_8x8x32_1x1_int8_1               nvdla_utb           PASS
pdp_7x9x10_3x3_int8                 nvdla_utb           PASS
sdp_8x8x32_bypass_int8_1            nvdla_utb           PASS
sdp_8x8x32_bypass_int8_0            nvdla_utb           PASS
sdp_4x22x42_bypass_int8             nvdla_utb           RUNNING
cdp_8x8x32_lrn3_int8_1              nvdla_utb           RUNNING
cdp_8x8x64_lrn9_int8                nvdla_utb           RUNNING
dc_24x33x55_5x5x55x25_int8_0        nvdla_utb           RUNNING
-------------------------------------------------------------------------------------------------
```

Wait for several minutes, all tests will be passed, and the final output will be

```
[INFO] Dir = TOT/health_exam/protection_tests
-----------------------------------------------------------------------------------------------------
Test                                    TB                  Status     Errinfo
-----------------------------------------------------------------------------------------------------
pdp_8x8x32_1x1_int8_1                   nvdla_utb           PASS
pdp_7x9x10_3x3_int8                     nvdla_utb           PASS
sdp_8x8x32_bypass_int8_1                nvdla_utb           PASS
sdp_8x8x32_bypass_int8_0                nvdla_utb           PASS
sdp_4x22x42_bypass_int8                 nvdla_utb           PASS
cdp_8x8x32_lrn3_int8_1                  nvdla_utb           PASS
cdp_8x8x64_lrn9_int8                    nvdla_utb           PASS
dc_24x33x55_5x5x55x25_int8_0            nvdla_utb           PASS
-----------------------------------------------------------------------------------------------------
TOTAL    PASS      FAILED    RUNNING   PENDING   Passng Rate
8        8         0         0         0         100.00%
```

Simulation log and result could be found under `TOT/health_exam/protection_tests/nvdla_utb` :

```
protection_tests
`-- nvdla_utb
    |-- cdp_8x8x32_lrn3_int8_1
    |-- cdp_8x8x64_lrn9_int8
    |-- dc_24x33x55_5x5x55x25_int8_0
    |-- pdp_7x9x10_3x3_int8
    |-- pdp_8x8x32_1x1_int8_1
    |-- sdp_4x22x42_bypass_int8
    |-- sdp_8x8x32_bypass_int8_0
    `-- sdp_8x8x32_bypass_int8_1
```

Verification Suite

NVDLA verification suite contains test bench, test plan and test suites.

Verification related files could be found under `TOT/verif` :

```
verif
|-- regression
|   `-- testplans
|-- testbench
|   |-- trace_generator
|   `-- trace_player
|--<some other directories>
`-- tests
    |-- trace_tests
    `-- uvm_tests
```

Test Bench Architecture

NVDLA verification adopts coverage driven methodology which relies on constrained random stimulus.

Test bench also need to support direct tests for other considerations which require fixed invariant stimulus:

1. Protection tests for code commit quality check.
2. Sanity tests for fast flushing plain and simple bugs.
3. Tests from real application (real network).

Simulation flow is divided into two phases: stimulus recording (trace generation) and stimulus playing (trace playing). There are independent executables for different phases.

1. Trace generation: a trace generator response for generating traces from constrained random tests. It contains with random constraints and random test suite.
2. Trace playing: a trace player response for driving trace to DUT, checking DUT correct behavior, and collecting coverages. It contains with agents for interacting with DUT, reference mode for correct behavior checking and coverage model for verification completeness measurement.

Trace generator

To be done

Trace player

To be done

Test Plan

Test plan record tests and corresponding configurations for regression. There is one test plan for one configuration project. Test plan file location is: `TOT/verif/regression/testplans` .

Currently, this only one valid test plan: `NV_SMALL` .

Test Levels

One test plan contains several test levels:

- Level 0: Pass through cases which are based read data from memory and write to memory without function operation. Basic function tests. Most of protection tests will be select from this level.
- Level 1: Common function case. Basic features such as major ASIC data path, special memory alignments, tests are single layer tests.
- Level 2: Corner cases, for example, extreme small cube size (1x1x1 in width,height,channel), maximum width cube size (8192x1x1 in width,height,channel).
- Level 3: reserved for multi-layer cases, run multiple layers on the same hardware pipelines.
- Level 4: reserved for multi-scenario, running independent hardware pipelines (convolution, pdp, cdp) in the same time.
- Level 5: reserved for perf tests
- Level 6: reserved for power tests
- Level 7: reserved for time-consumed tests
- Level 8: reserved for real network cases
- Level 9: reserved
- Level 10: random tests for single scenario
- Level 11: random tests for multi scenario

Level 0 to 8 are considered as direct tests, level 10 to 11 are considered as random tests. There are dedicated test list files for each test level.

Associating Tests

Test plan provides a method named add_test to associate tests with test plan, in each test, there are 5 fields:

- Name: test source name
- Tags: tags for test selection
- Args: required arguments for test simulation, arguments will be documented in test bench and
- Config: target test bench, currently, there is only one valid test bench setting: nvdla_utb which stands for NVDLA Unit Test bench.
- Module: use to distinguish different types of tests
- Desc: test description

Example:

```
add_test(name='dc_24x33x55_5x5x55x25_int8_0',
         tags=['L0','cc','protection'],
         args=[FIXED_SEED_ARG, DISABLE_COMPARE_ALL_UNITS_SB_ARG],
         config=['nvdla_utb'],
         desc='''copied from cc_small_full_feature_5, kernel stride 4x3, unpacked, pad L/R/T/B, clip t
```

Test Suite

There are two kinds of tests:

1. Direct tests: direct tests are in trace format. Test source files are under TOT/verif/tests/trace. Trace tests are organized by configuration project. Trace tests for NV_SMALL configuration is under `TOT/verif/tests/trace_tests/nv_small` .
2. Random tests: random tests are in UVM test format. Random tests are expected to be configuration independent. Random test directory is `TOT/verif/tests/trace_tests/uvm_tests` .

**For now, only trace tests are ready.*

Test naming

Test naming is convenience for fast understanding test scenarios. There are several portions in test name:

<MAJOR_FUNCTIONS>_<DIMEMSION_SETTINGS>_<SUPPLEMENTARY_INFO>_<MINOR_FUNCTIONS>_<PRECISION>_<INDEX>

Portion Explanation

Portion	Description	Necessity
MAJOR_FUNCTIONS	Major hardware pipelines. Possible values: • DC: direct convolution, input is feature cube format • IMG: direct convolution, input is image format • WINO: Winograd convolution, input is feature cube format	Must have
DIMEMSION_SETTINGS	Cube dimensions, for convolution tests, there are two cubes, one is for data, one is for weight. Dimension orders. • For data: width, height, channel • For weight: width, height, channel, kernel	Must have
SUPPLEMENTARY_INFO	Supplementary information, for examples: • In image convolution, image format • In SDP, there are multiple operation units like BS/BN. • In PDP, pooling stride settings	Optional
MINOR_FUNCTIONS	Minor functions within major hardware pipeline. For example, in convolution pipeline, there are weight compression, dilation etc.	Optional
PRECISION	Specify working precision.	Must have

Portion	Description	Necessity
INDEX	Some tests share the same configuration but different memory surface, use index to distinguish those tests	Optional

Examples

Test name	Scenario description
dc_24x33x55_5x5x55x25_int8_0	direct convolution, input feature cube dimension is 24x33x55 (width, height, channel), weight cube dimension is 5x5x55x25 (width, height, channel, kernel), precision is INT8
sdp_3x3x33_bs_int8_reg_0	Single data processing, input feature cube dimension is 3x3x33 (width, height, channel), using BS operation unit, operand from register, precision is INT8
pdp_8x8x64_2x2_int8	Pooling, input feature cube dimension is 8x8x64 (width, height, channel), precision is INT8

Test Format

Trace Test format

In unit test bench, trace player only receives tests in trace format. Trace test is the source format of direct tests and the inter-media outputs of random tests.

Trace test is not only used in unit RTL verification environment, but also could be reused in other environments like system verification, CMOD verification, FPGA validation, and silicon bringup.

Trace format supports following cases:

1. Single layer case: only one hardware pipeline and only one configuration group will be exercised during simulation.
2. Multi-layer case: configuration group will be alternative used during simulation within the same hardware pipeline.
3. Multi-resource cases: one hardware pipeline which consists of multiple hardware resource. For example, a fused convolution-batch_normlization-relu-pooling layer could be executed in single hardware layer, this hardware layer requires several computational resources convolution, SDP and PDP.
4. Multi-scenario case: multiple independent hardware layer configuration could be presented in the same test.

Trace stores the stimulus of simulation. There are two types of stimulus, and there is dedicated format to support each type of stimulus:

1. Register configuration, stored in config file (file name is *.cfg). One test only has one configuration file.
2. Memory data, stored in data files (file name is *.dat). One test has at least one memory data file for input data. One test could have more than one data file, for example, in convolution tests, there is one file for input image/feature cube and one file for weight.

Trace file also provide result checking methods:

1. Golden CRC, which is represented in one configuration command.
2. Golden output memory surface, which is represented in one configuration command and an associated data file.

Configuration File Format

There are 9 types of configuration commands:

Name	Description	Syntax	Use case
reg_write	Write data to specific DUT register	reg_write(reg_n ame, reg_value);	Fundamental operation for register configuration
reg_read_expect ed	Read data from specific DUT register, compare with expected value	reg_read_expect ed(addr, expected_data);	For some special cases like register accessing tests

Name	Description	Syntax	Use case
reg_read	Read data from specific DUT register	reg_read(reg_na me, return_value);	For specific cases which may need to do post-processing on read return value.
sync_notify	Specified player sequencer will send out synchronization event	sync_notify(tar get_resource, sync_id);	CC pipeline, OP_EN configuration order, CACC->CMAC->CSC .
sync_wait	Specified player sequencer will wait on synchronization event	sync_wait(targe t_resource, sync_id);	CC pipeline, OP_EN configuration order, CACC->CMAC->CSC .
intr_notify	Monitor DUT interrupt, catch and clear interrupt and send synchronization event. There could be multiple intr_notify, all those intr_notify are processed sequentially. The processing order is the same as commands' line order in configuration file.	intr_notify(int r_id, sync_id); // notify when specific interrupt fired	Hardware layer complete notification, informing test bench that test is ended. Multi-layer test which is presumed containing layer 0 ~ N, for n >1 layers, they shall wait for interrupts.
poll	Continues poll register/field value from DUT, until one of the following conditions are met: 1. Equal, polled value is equal to expected value 2. Greater, polled value is greater than expected value 3. Less, polled value is less than expected value 4. Not equal, polled value is not equal to expected value 5. Not greater, polled value is not greater than expected value 6. Not less, polled value is not less than expected value	poll_field_equa l(target_resour ce, register_name, field_name, expected_value) ; poll_reg_equal(target_resource , register_name, expected_value) ; poll_field_grea ter(target_reso urce, register_name, field_name, expected_value) ; poll_reg_less(t arget_resource, register_name, expected_value) ; poll_field_nt_ greater(taget_ resource, register_name, field_name, expected_value) ; poll_reg_not_le ss(target_resou rce, register_name, expected_value) ;	Convolution case, wait until CBUF flush has done

Name	Description	Syntax	Use case

check	Invoke player result checking method.
	
	When test bench works in RTL/CMOD cross checking mode, neither golden CRC nor golden files are necessary in this case. Method check_nothing() shall be added to trace file to indicated test end event.
		check_crc(syn_ id, memory_type, base_address, size, golden_crc_valu e);
		
		check_file(sync _id, memory_type, base_address, size, “golden_file_na me”);
		
		check_nothing(s ync_id);
			CRC check for no CMOD simulation (usually generated by arch/inherit from previous project/eyeball gilded)
			
			Golden memory result check for no CMOD simulation (usually generated by arch/inherit from previous project/eyeball gilded)

mem	Load memory from file.
	
	Initialize memory by pattern.
		mem_load(ram_ty pe, base_addr, file_path); // file_path shall be enclosed by “”
		
		mem_init(ram_ty pe, base_addr, size, pattern);	

*Some functions are not supported yet.

Memory Surface File Format

When mem_load command is shown in configuration file, test bench will load corresponding file into memory model.

Memory surface format is in memory mapped form. The basic data group is call packet, each packet item contains one address offset field, one size field and one payload field. It’s used to store data in payload field to memory space starting from address (base + offset).

The string describing one packet item must be kept in one single line.

Payload byte must be separated by space, and payload size shall not be no larger than 32 for readability consideration.

Different packets shall be joined by comma “,”.

Example:

```
{

{offset:0x20, size:4, payload:0x00 0x10 0x20 0x30} ,

{offset:0x60, size:4, payload:0x00 0x10 0x20 0x30}

}
```

For packet `{offset:0x20, size:4, payload:0x00 0x10 0x20 0x30}` , data in memory layout is

Address	0x20	0x21	0x22	0x23
Value	0x00	0x10	0x20	0x30

Random Test Format

To be done

Regression Tool Set

There is a tool set for:

1. Running single test simulation.

2. Running a test plan, tests associated with specific test plan will be running. It could be considered as one round of regression.
3. Examining single round regression result and generate metrics for whole project lasting time.

Running a Single Test

There is a script TOT/verif/tools/run_test.py for running single test, the most common usages are:

```
>TOT/verif/tools/run_test.py -P <project_name> -mod <test_module> <trace_test_name> -outdir
<output_directory> -v nvdla_utb
```

Argument Explanations:

Argument	Description
-P <project_name>	Project name which is specified in tree.make
-mod <test_module>	Specifying test kind, if it is not specified, will select trace test by default
<trace_test_name>	Test name which could be found under project related trace directory
-outdir <output_directory>	Specifying working directory, temporal files and log will be generated in output_directory, if outdir has not been specified, current directory will be used as working directory.
-v nvdla_utb	Specifying test bench, for now, only unit test bench is available, so nvdla_utb is the only valid argument

Please run `run_test.py` with `-help` argument for more arguments and their usages.

Examining Result

There are several files were generated during simulation, three files need to be paid more attention:

1. run_trace_player.sh: a script for rerunning test
2. run_verdi.sh: a script for kicking off Verdi to view waveforms
3. testout, log file
4. STATUS, file records test running status, there are several status:

    1. RUNNING, test is still in running.
    2. FAIL, test result is failure.
    3. PASS, test result is pass.

Running a Test Plan

There is a script TOT/verif/tools/run_plan.py for running tests within a test plan.

```
>TOT/verif/tools/run_plan.py --test_plan <TEST_PLAN_NAME> -P <PROJECT> -atag <and_tags> -otag <or_tags> -
run_dir <RUN_DIR> -no_lsf -monitor
```

Argument Explanations:

Argument	Description
–test_plan <TEST_PLAN_NAME>	Test plan file name, without '.py' suffix.
–test_plan <TEST_PLAN_NAME>	Project name which has been specified in tree.make
-atag <and_tags>	means AND tags, will select tests which has all tags specified by atag
-otag <or_tags>	means OR tags, will select tests which contain at least one of tags specified by otag
-ntag <not_tags>	means NOT tags, will select tests which don't not contain any tags specified by ntag

Argument	Description
-run_dir <RUN_DIR>	Specify working directory
-no_lsf	will run test on local CPU
-monitor	will continuously monitoring test running status, until all simulations are done or reach maximum runtime

Please run `run_plan.py` with `-help` argument for more arguments and their usages.

Simulation results could be found under **run_dir**. There are 2 hierarchy levels under `run_dir`

```
RUN_DIR
|-- <TEST_BENCH_0>
|   |-- TEST_A
|   |-- TEST_B
|   |-- …
|   `-- TEST_G
`-- <TEST_BENCH_1>
    |-- TEST_H
    |-- …
    `-- TEST_N
```

1. The first level is test bench level. If a plan contains multiple test benches, there will be dedicated directory for each test bench
2. The second level is test level. Under test bench level, there are several test directories. Each directory contains temporal and result files for one test simulation.

Examining result

There is a script TOT/verif/tools/run_report.py for monitoring regression status, the most common usages are:

>TOT/verif/tools/run_report.py -run_dir <regression_run_directory> -monitor_timeout MONITOR_TIMEOUT -monitor

Please run run_test.py with -help argument for more arguments and their usages.

Generating Reporting Metrics

To be done

Here is the end of **NVDLA Verification Suite User Guide**.