howto Documentation

Release 1

Germain Haugou

February 20, 2017

CHAPTER

ONE

WHAT MUST BE CONFIGURED BEFORE USING OR COMPILING THE SDK

Artifactory access

Toolchains and SDK releases are deployed on the artifactory server in order to quickly get prebuilt versions without having to recompile them.

Before accessing the server, the user and password must be properly configured with the following environment variable:

```
$ export PULP_ARTIFACTORY_SERVER=pulp:<password>
```

The password can be retrieved from the pulp_pipeline top module sources (at git@iis-git.ee.ethz.ch:pulp-sw/pulp_pipeline.git):

```
$ cat bin/ci.py | grep "pulp:"
```

When the SDK is retrieved through the downloader, it is downloading using wget without any user / password setting, thus the best way is to add a file .wgetrc in your home folder containing the following:

```
user=pulp
password=<password as above>
```

Linux distribution

The SDK should work on Ubuntu 14 or 15, Centos 6 or 7 and Linux Mint.

The binary packages retrieved from the artifactory server are compiled for a specific Linux distribution and so you must get the appropriate one. The packages are currently compiled for Ubuntu 14 and Centos 6 and as they are compatible all the Linux and Mint distributions are mapped to Ubuntu 14 and all Centos distributions are mapped to CentOS 6.

This will differ in the future as for example some Ubuntu versions won't be compatible together. When this happens, there will be an automatic mapping to the right distribution that should work most of the time. In case it does not work and you want to force the version of the packages, you can define the following environment variable:

```
$ export PULP_DISTRIB=<package distrib>
```

The possible values are curently Ubuntu_14 or CentOS_6.

System packages

Here are the list of packages that you have to install starting from a fresh Ubuntu 15.10 distribution:

```
$ sudo apt-get install git g++ python3-setuptools swig3.0 python3-dev libjpeg-dev libz-dev cmake lib3
$ sudo easy_install3 artifactory jenkinsapi sqlalchemy prettytable
```

The following packages are also needed to fully compile the SDK:

```
$ sudo apt-get install python-sphinx man-db pexpect texlive-latex-base texlive-latex-extra doxygen
```

CHAPTER

TWO

HOW TO USE PLATFORM TRACES

Platform traces are information dumped by the platform in order to help the developper debugging his application. These traces correspond to HW events like an executed instruction and thus give details about what is happenning in the system. Although these traces are available on several platforms (currently virtual platform and RTL platform), the supported events can differ.

How to activate them

On the RTL platform, the traces are always active and can be found in the test build folder (the one given to pulp-run through option *–dir*). As it only supports core instruction traces, there is one file per core with the name trace_core_<clusterId>_<coreId>.log.

On the virtual platform, they are by default not active and can be activated and configured through the runner option *–gv-trace*. This option can be either passed directly to the *pulp-run* command or through the makefile using:

```
$ make run PLT_OPT=--gv-trace=.*
```

The traces are also generated in the test build folder. As it is dumping not only core instruction traces but also system-wide traces, there is a single file called *trace.txt*, which is generated and contains everything.

How to configure them

On RTL platform, the traces cannot be configured as only core instruction traces are always generated. See below for more information about these traces.

The virtual platform allows dumping all kind of architecture events to help the developper debugging his application by better showing what is happening in the system.

For example, it can show instructions being executed, DMA transfers, events generated, memory accesses and so on.

This feature can be configured through the *–gv-trace* option. This option takes an argument which specifies the path in the architecture where the traces must be enabled. All components included in this path will dump traces. Each path is a regular expression, so it can be partial paths, use wildcards and so on. Several paths can be specified by using several times the option. Here is an example that activates instruction traces for core 0 and core 1:

```
$ make run PLT_OPT="--gv-trace=pe0/iss --gv-trace=pe1/iss"
```

One difficulty is usually to find out which paths should be activated to get the needed information. One method is to dump all the events with *–gv-trace=.**, then find out which one are insteresting and then put them on the command line. Here are the paths for the main components:

Path	Description
/soc/fabric/cluster0/pe0	Processing element, useful to see the IOs made by the core, and the instruction it executes. You can add */iss* to just get instruction events
/soc/fabric/cluster0/ckg	Hardware synchronizer events, useful for debugging inter-core synchronization mechanisms
/soc/fabric/cluster0/pcache	Shared program cache accesses
/soc/fabric/cluster0/l1ico	Shared L1 interconnect
/soc/fabric/cluster0/l1_X	L1 memory banks (the X should be replaced by the bank number)
/soc/fabric/l2	L2 memory accesses
/soc/fabric/cluster0/dma	DMA events

Traces content

The trace file should look like the following:

```
20830000: [/soc/fabric/cluster0/pe1/iss    ]      1c0001f0 l.lwz              r6 , 540(r6 )     r6 =
```

There is usually one line per event, although an event can sometimes takes several lines to display more information.

The number on the left gives the timestamp of the event, in picoseconds. This is not using cycles because different blocks like clusters can have different frequencies.

The second part, which is a string, gives the path in the architecture where the event occured. This is useful to differentiate blocks of the same kind that generate the same event. This path can also be used with the *–gv-trace* option to reduce the number of events.

The third part, which is also a string, is the information dumped by the event, and is totally specific to this event. In our example, the core simulator is just printing information about the instruction that has been executed.

Core instruction traces

Core instruction traces are available on both the RTL platform and the virtual platform. As they are showing not only the instructions executed but also the registers accessed, their content and the memory accesses, they are very useful for debugging bugs like memory corruptions.

If we take back the previous example:

```
20830000: [/soc/fabric/cluster0/pe1/iss    ]      1c0001f0 l.lwz              r6 , 540(r6 )     r6 =
```

The event information dumped for executed instructions is using the following format:

```
<address> <instruction> <operands> <operands info>
```

<address> is the address of the instruction.

<instruction> is the instruction label.

<operands> is the part of the decoded operands.

<operands info> is giving details about the operands values and how they are used.

The latter information is using the following convention:

- When a register is accessed, its name is displayed followed by = if it is written or *:* if it is read. In case it is read and written, the register appears twice. It is then following by its value, which is the new one in case it is written.
- When a memory access is done, *PA:* is displayed, followed by the address of the access.

- The order of the statements is following the order on the decoded instruction

The memory accesses which are displayed are particularly interesting for tracking memory corruptions as they can be used to look for accesses to specific locations.

CHAPTER

THREE

HOW TO USE VIRTUAL PLATFORM VCD TRACES

The virtual platform can dump VCD traces which show the state of several components over the time, like the cores PC, the DMA transfers, etc.

Configuration

Everything can be activated with this single option:

```
make run PLT_OPT=--vcd
```

This will activate VCD traces and also all possible traces and will dump them in the platform working directory in the file called *all.vcd*.

It is also possible to select exactly which traces must be activated with this option:

```
make run PLT_OPT="--vcd --vcd-file=pc.vcd:.*/pc"
```

The *–vcd* option is still required to activate VCD traces. However as there is the *–vcd-trace* option, only the specified traces are activated.

The option must follow the format –vcd-file=<traceFile>:<tracePattern>. <traceFile> is the path of the file where the specified traces will be dumped. <tracePattern> is a Posix regular expression which specifies the traces that must be dumped to this file. Each VCD trace has a path made of the concatenation of the path of the component it belongs to and the trace name. Any trace whose path matches the regular expression will be dumped to the specified file.

It is also possible to specify several files in case one wants to separate VCD traces:

```
make run PLT_OPT="--vcd --vcd-file=pe0.vcd:.*/pe0/iss --vcd-file=pe1.vcd:.*/pe1/iss"
```

This will in this case dump PE0 and PE1 PC traces to 2 different files.

Display

Any VCD viewer can be used to display the traces. On Linux the free gtkwave viewer can be used. For example to display the PC traces, you can launch it with:

```
gtkwave <vcdFilePath>
```

Then click on Search->Search Signal Regexp, enter “pc”, click on Select All and Insert, and close the box. You should now see the PC traces in the view, you can zoom out to see the full window.

Gtkwave script generation

In case gtkwave is used, a script can be automatically generated that will preconfigure all interesting signals. For that the following option must be used:

```
$ make run PLT_OPT="--vcd --gtkw"
```

You should se at the beginning of the simulation that says that a script has been generated. Just open it with gtkwave and you should see all signals already configured in the view.

This view contains some pre-defined groups that are very useful for quickly opening and closing architectures parts. It also contains a group called debug, that contains more high-level debug information like function name, disasembled instructions and so on.

CHAPTER

FOUR

HOW TO DEBUG MEMORY CORRUPTIONS OR INVALID ACCESSES

Note that this howto can only be used on the virtual platform and is strongly relying on platform traces described here: *How to use platform traces*.

First identify the reason of the memory corruption or the invalid access

Such bugs can show-up in different ways.

The first one is that an invalid access is reported by the platform like the following:

```
58988800000: [/soc/fabric/noc/warning               ] Invalid access (offset: 0x3f14fc00, size: 0x10,
58988800000: [/soc/fabric/cluster0/pe0/warning     ] Invalid access during instruction fetch (offset:
```

These ones are the easiest to debug as they are explicitly reported by the platform, as some kind of asserts. In this case, there was an invalid access reported by the interconnect (in the architecture at */soc/fabric/noc*) which ends up being reported by the core doing the access as an invalid instruction fetch.

They can also be reported by other warnings reported by the platform like invalid use of peripherals or registers.

Unfortunatly most of the bugs will just lead to invalid results or hanging application. In this case, the application should be debugged using printf of gdb (soon available).

Get more information using platform traces

In case a platform warning is raised, the following method can be applied to get more information.

As the platform is stopped as soon as the invalid access is raised (by default warnings are considered as errors), it is really easy to have more information by dumping the end of the executed instructions using platform traces:

```
58988690000: [ESC[34m/soc/fabric/cluster0/pe0/iss         ESC[0m] 1c003c5c l.jr                 r9
58988700000: [ESC[34m/soc/fabric/cluster0/pe0/iss         ESC[0m] 1c003c60 l.lwz                r30, 
58988800000: [ESC[34m/soc/fabric/cluster0/pe0/iss         ESC[0m] 3f14fc0c l.j                  0
```

We see here that the core is jumping to an invalid address (0x3f14fc0c), which is also more or less the one reported as being the address of the invalid access. This faulting address is a bit different as it is actually a program cache line refill. The jump to this invalid address is done because the return register (r9) is containing this invalid address.

The next step is to find out how we ended-up having this invalid address in r9, which can be done by looking for r9 in the platform traces from the end to the beginning. If for example you open the trace file with the tool *less*, you can look for it by typing /, followed by r9 and enter, and then shift + n to go backward or n to go forward. Doing that we see that r9 is written by the following operation:

```
58988420000: [ESC[34m/soc/fabric/cluster0/pe0/iss         ESC[0m] 1c003c38 l.lwz                   r9 , -
```

This operation is loading the invalid address from memory at address 0x3420 (which is an L1 address through the cluster alias), and storing it to r9. As you can see on the operands part, the address is computed from r1 which is the stack pointer. Probably the address was stored on the stack earlier (e.g. before calling a function) and is now being restored in order to return to the caller (r9 is the return address). In such cases, most of the time, another unrelated code stored something else at the same location illegaly, for example with a buffer overflow.

The next step is to find out which code is storing this invalid address on top of the valid one which was saved. To do that, just continue to search in the platform traces from the end to the start by looking for *PA:00003420*. Each time a core is accessing this address, this pattern will appear on the line containing the instruction doing the access. In this example, the following instruction is doing the access:

```
58945820000: [ESC[34m/soc/fabric/cluster0/pe0/iss         ESC[0m] 1c0038b0 l.sw                    8(r6
```

This instruction is very suspicious, first because it stores something in a location used to save registers on the stack, and is not computing the address using the stack pointer (it is using r6), and also because it is storing a value at a location where a pointer is expected.

To know why is the code doing this, just disassemble the application binary (using or1kle-elf-objdump -d -l or riscv32-elf-objdump -d -l) and look for address 0x1c0038b0 (the address of the instruction), this should give the source code file and the line of the code doing that. In this example, this points out to a code which was writing to this address due to a buffer overflow (an array was too small) and then overwriting something on the stack as the array was on the stack.

Get more information using platform debugger

If you look at the pulp-run command launched to start the execution, you should see a –pdb-no-break, which tells the platform to not enter the debugger at start-up. If you replace it with –pdb-break, the platform should first enter the debugger before executing the first instruction and you should get the following prompt:

```
(Cmd)
```

You can use the bkp command to register a breakpoint on a memory address so that the platform stop in case this address is accessed. For that just enters:

```
(Cmd) bkp 0x3420
(Cmd) run
```

This will run the platform and you should see something like this when the address is accessed:

```
403350000: [/soc/fabric/cluster0/pe0/userTrace   ] Reached breakpoint (matchAddr: 0x00003420, matchAd
(Cmd)
```

In case this is not the access you are looking for, you can retype the run command or execute the following to get the PC of all cores:

```
(Cmd) state
/soc/fabric/cluster0/pe0
  PC: 0x1c001ed4
/soc/fabric/cluster0/pe1
  PC: 0x1c004b0c
/soc/fabric/cluster0/pe2
  PC: 0x1c004b0c
/soc/fabric/cluster0/pe3
  PC: 0x1c004b0c
```

You can then use the PC information and objdump to know which source code is doing this access.

Get more information using GDB

GDB should soon be available for tracking memory accesses using watchpoints.

CHAPTER

FIVE

HOW TO USE GDB WITH VIRTUAL PLATFORM

The virtual platform can open a debug bridge so that GDB can connect on it.

Usage

The virtual platform must be launched with the following option:

```
make run PLT_OPT=--debug
```

This should dump something like this:

```
Debug bridge listening on port 1234
```

Then the platform is stopped (all core are in debug mode), waiting for gdb to connect.

In another terminal, gdb must be launched with:

```
riscv32-unknown-elf-gdb
```

Then the following command must be executed from gdb shell:

```
target remote :1234
```

If everything runs fine, this should display:

```
0x1c000080 in ?? ()
```

This means gdb sees that the current selected core is stopped at the entry point (this is the address of the reset exception handler).

To see which cores are connected:

```
info threads
```

This should display:

```
  Id   Target Id         Frame
* 1    Remote target     0x1c000094 in ?? ()
  2    Thread 1 (Core 00000001) 0x1c000094 in ?? ()
  3    Thread 2 (Core 00000002) 0x1c000094 in ?? ()
  4    Thread 3 (Core 00000003) 0x1c000094 in ?? ()
```

What is supported

Everything should be supported, including instruction-stepping, breakpoints, watchpoints, etc. The cores are configured in a symetric mode where all cores stop as soon as at least one stops. This means, if one core reaches a breakpoint, all cores will be stopped as if they have also reached a breakpoint.

However the gdb porting to riscv is not fully stable and is thus generating lots of invalid accesses, especially when unwinding the stack.

Tips

The socket port where gdb must connect is by default 1234 but can be specified with the following option if for example it is in conflict with another application:

```
make run PLT_OPT="--debug --gdb-port=12345"
```

The gdb target command execute to connect to the platform must give the same port.

CHAPTER

SIX

HOW TO USE RUNTIME TRACES

Runtime traces are information dumped on the console by the runtime in order to help the developper debugging his application. These traces are dumped when there is an interesting event that happends in the runtime. These trace are only available with the new runtime (option pulpOmpVersion=new).

How to activate them

CHAPTER

SEVEN

HOW TO USE THE RTL PLATFORM

The RTL platform is integrated inside the SDK runner which allows launching it from the usual make command for running the application.

Before using it, it must be properly built and configured, which can be done in 2 ways.

Using the SDK flow

The main advantages is that it is fully integrated. However not all the chips are integrated, in this case, you can switch to the other way.

First you have to build the RTL following these instructions *How to build the RTL platform*.

Then simply execute this:

```
$ make clean all run
```

Note that this will work only if the SDK is configured for the RTL platform using for example:

```
$ plpconf --platform=rtl
```

Each chip has its own install folder so you can easily switch from one chip to another automatically using different SDK configurations and without recompiling the RTL platform.

Using an external RTL platform

In case your RTL platform does not fit well the SDK flow, you can still build it on your own, the way you want. Just make sure that the environment variable PULP_RTL_INSTALL is pointing to the RTL platform, to the folder containing modelsim.ini.

Then you can rely on the classic SDK flow to compile and run the application (see above).

Tips

You can launch the RTL platform using the GUI with this option:

```
$ make clean all run gui=1
```

You can recompile the application and keep the GUI opened, just reset it, and you can run again the test.

In case you have to do things by hands, the vsim command used to launch the platform is printed at the beginning. You can launch it by hands from the build folder, which you can also find on the pulp-run command that is printed. All the files needed to launch the platform, like the stimuli, are inside the build folder.

CHAPTER

EIGHT

HOW TO BUILD THE RTL PLATFORM

Outside the SDK flow

Just compile it the way you want and define the environment variable PULP_RTL_INSTALL to the install folder, the one containing modelsim.ini. Then you can use it using the SDK flow.

Using the SDK flow

First configure the SDK with the proper configuration, for example with:

```
$ plpconf --platform=rtl --pulpChip=pulp4
$ source setup.sh
```

Then you can either execute that from your application:

```
$ make rtl VERBOSE=1
```

Or from anywhere:

```
$ plpbuild --p rtl --stdout checkout build
```

This command will checkout and compile the RTL platform that corresponds to your configuration, in this case pulp4.

Once the build step has succeeded, you can compile and run an application using the classic command:

```
$ make clean all run
```

Tips

Be careful, when you execute “make rtl” this checkout specific sources of the platform. This can mess-up your sources in case you have modified something or manually switched to a specific version.

In case you are modifying the platform and don’t want to checkout new sources, you can execute:

```
$ make rtl.build VERBOSE=1
```

Or:

```
$ plpbuild --p rtl --stdout build
```

In case you want to compile only a few ips:

```
$ make rtl.build VERBOSE=1 BUILD_OPT="--ip mchan --ip udma --ip udma_subsystem"
```

Or:

```
$ plpbuild --p rtl --stdout build --ip mchan --ip udma --ip udma_subsystem
```

To compile the whole RTL or just an IP, the tool is just relying on the Makefile at the top of the RTL repository, which decides at the end what has to be launched depending on the options. In case there is an issue, the Makefile may not be up-to-date with the way the RTL must be compiled.

CHAPTER

NINE

HOW TO SETUP SDK ENVIRONMENT FOR RTL DEVELOPMENT

This howto is targetting people who are more on the HW side and wants to use the exact SDK that must be used with on-going developments on RTL side. Note that it is supported only since pulp-dev and fulmine.

SDK setup

Before accessing the server, make sure artifactory access is setup using this doc: *What must be configured before using or compiling the SDK*

Compile the RTL

The RTL can be compiled the usual way. For example, for pulp-dev, first checkout the repository:

```
$ git clone git@iis-git.ee.ethz.ch:pulp-project/pulp-dev.git
$ cd pulp-dev
$ ./update-ips.py
```

And then compile the RTL platform:

```
$ cd fe/sim
$ source vcompile/setup.csh
$ ./vcompile/build_rtl_sim.csh
```

Get the SDK

The following script can be executed in order to get the SDK version that must be used with this version of the RTL, checkout the SDK sources and compile them (execute it from the top RTL directory):

```
$ ./update-sw
```

Note that there is currently a bug with NFS which makes the build fails with strange errors like “make: r: Command not found”. The work-around until it is fixed, is to reexecute the script.

Setup the SDK

Once the SDK has been checked-out and compiled, the terminal must be configured by sourcing the following file:

```
$ source setup/sdk.csh
```

Note that this file must always be sourced before using the SDK, so that the SDK knows where to find the RTL platform.

Get and run the tests

The tests can be checked-out with the followin command:

```
$ ./update-tests
```

You can have a look here to have information about the tests: *Where to put a new test or application*

You can then compile and run a test with:

```
$ cd pulp_pipeline/tests/pulp_tests/helloworld
$ make clean all run
```

Run a testsuite

A testsuite can be launched using plptest from any level in the hierarchy of tests. For example all mchan tests can be launched by executing:

```
$ cd pulp_pipeline/tests/pulp_tests/testMCHAN/v5
$ plptest
```

Or they can be launched or all pulp_tests with:

```
$ cd pulp_pipeline/tests/pulp_tests
$ plptest
```

To launch the full set of tests:

```
$ cd pulp_pipeline
$ plptest
```

Add a test

You can have a look here for adding a test: *How to write a test*

CHAPTER

TEN

WHERE TO PUT A NEW TEST OR APPLICATION

Organization

The Pulp tests and applications are currently spread in the following git repositories:

- **OR10N tests**: git@iis-git.ee.ethz.ch:pulp-sw/or10n_tests.git

  These tests can only be used with the old OR10N core and should not be modified anymore. They are kept in order to still be able to launch tests on old chips like Mia Wallace, for example to validate the virtual platform.

- **RISCV tests**: git@iis-git.ee.ethz.ch:pulp-sw/riscv_tests.git

  These tests can only be used with the first RI5CY version (named ri5cyv1 in the SDK) and should not be modified anymore. They are kept in order to still be able to launch tests on old chips, for example to validate the virtual platform.

- **Core tests**: git@iis-git.ee.ethz.ch:pulp-sw/core_tests

  These tests are the current ones used for validating the latest OR10N (or10nv2) and RISCV (ri5cyv2) cores. Any test focusing on core validation should be there. The whole testsuite is supposed to support both cores, although some tests are specified to run only on one core. A test that runs on both cores is using macros to be compilable for both cores.

- **Pulp RTL tests**: git@iis-git.ee.ethz.ch:pulp-sw/pulp_tests.git

  The tests are used to validate specific features of the RTL, except for the core which has its own testsuite. There are for examples tests for the DMA, the UDMA, etc. The tests are supposed to work on all chips, except if the compatibility is broken. In this case, the old tests should be kept so that the testsuite can still be run on older chips. For example the mchan tests are duplicated in a v4 and v5 subdirectories in order to work on latest chips.

- **Sequential bare-metal tests**: git@iis-git.ee.ethz.ch:pulp-sw/sequential_bare_tests.git

  These are single-core tests mainly useful for benchmarking the core or the instruction cache. They were initially used to benchmark the ISA extensions. They are also supposed to work on any chip and should not be duplicated as they are quite generic.

- **Parallel bare-metal tests**: git@iis-git.ee.ethz.ch:pulp-sw/parallel_bare_tests.git

  These are multi-core tests useful for benchmarking parallel aspects of the architecture. They use very basic synchronization (just a barrier) so they should run on any chip without any problem.

- **OpenMP tests and basic applications**: git@iis-git.ee.ethz.ch:pulp-apps/gomp_tests.git

  Tests or applications that should remain small to use them as basic non-regression testsuite for OpenMP

- **GCC OpenMP testsuite**: git@iis-git.ee.ethz.ch:pulp-apps/libgomp_testsuite.git

  Should not be modified, it is the testsuite used to validate GCC OpenMP frontend. Just modified the minimal to run it on Pulp

- **LLVM OpenMP testsuite**: git@iis-git.ee.ethz.ch:pulp-apps/openuh.git

  Should not be modified, it is the testsuite used to validate LLVM OpenMP frontend. Just modified the minimal to run it on Pulp

- **OpenMP EPCC benchmark**: git@iis-git.ee.ethz.ch:pulp-apps/epccbench.git

  Should not be modified, it is a well-known benchmark for OpenMP ported on Pulp.

- **OpenVX applications**: git@iis-git.ee.ethz.ch:pulp-ovx-rt/samples.git

  OpenVX applications used for benchmarking OpenVX dynamic graph execution.

- **OpenVX kernels tests**: git@iis-git.ee.ethz.ch:pulp-ovx-rt/ovx-kernels.git

  Tests for validing OpenVX kernels separatly. The repository also contains the kernels, the tests are under the tests directory. The tests are also good for basic OpenMP validation as they all using OMP parallel pragma to parallelize the work.

- **Coremark**: git@iis-git.ee.ethz.ch:pulp-apps/coremark.git

  The Coremark benchmark ported on Pulp. Usefull for quick validation of the core, as it executes various processing and control code, and also useful for giving a quick idea of the core performance, through a score.

- **Cconvnet tests**: git@iis-git.ee.ethz.ch:fconti/cconvnet.git

  Tests for the cconvnet library. The repository also contains the cconvnet library, the tests can be found under example.

- **Vlfeat tests**: git@iis-git.ee.ethz.ch:pulp-project/vlfeat.git

  Tests for the vlfeat porting on Pulp. The repository also contains the vlfeat library, the tests can be found under tests.

How to add a test

To add a test or an application, the appropriate repository can be cloned and the test added with the usual commands.

The whole tests and applications can also be retrieved using the SDK flow with this command:

```
$ plpbuild --p tests --p apps --p ovx_tests --p omp_tests checkout
```

Only a subset of the packages can be specified to not get everything. The *apps* package contains cconvnet and vlfeat, *ovx_tests* contains ovx_kernels and ovx_apps, *omp_tests* contains libgomp_testsuite, openuh and epccbench and *tests* contains the rest.

Currently all the tests are always pushed on the master branch as they are supposed to always work on all the chips.

CHAPTER ELEVEN

HOW TO WRITE A TEST

Get a working empty test

Once the SDK has been propertly configured, you can create an empty simple test by executing:

```
$ pulp-quickstart --path=test
```

This will create a helloworld in the test directory. This basically copy the example which is inside the SDK under examples/bare/hello.

Compile and run the test

You can go to the generated test folder and execute:

```
$ make clean all run
```

Configure the test properties for testsuite integration

The generated test contains a testset.ini file that allows launching this test using the plptest script that you can launch from the test folder:

```
$ plptest
```

This will execute the test as a non-regression testsuite which allows you checking if everything is fine for testsuite integration.

The test status is considered as passed if the test exits with an exit value of 0, otherwise it is considered as failed. This allows the test simply returning the C main entry point return value as the test status or using a script that can post-process the test output to determine if the test has passed or failed.

If you look at the testset.ini file, you will see the following:

```
[test:hello]
command.all=make clean all run systemConfig=%(config)s
timeout=1000000
```

The first line must just contain *test:* to indicate it is a single test followed by the test name that will appear in the report.

The second line is the command that is launch to compile and run the test. The command is here our classic *make clean all run* command followed by a property which is used by the test framework to propagate the SDK configuration for which the test must be validated. This for example allows launching the same test for several configurations, each configuration being caught by this property. This property should be kept in order to execute the test for the right configuration.

The third line is the timeout of the test, expressed in number of expected cycles. This timeout is used by a formula depending on the platform to compute the actual timeout in seconds, which allows expressing the same timeout for all platforms. The formula usually takes into account this number, the platform speed and the number of simulated core to estimate the timeout in seconds.

All these properties can most of the time be kept like that.

Integrate the test into a testsuite

A full non-regression testsuite is a hierarchy of testset.ini files putting all tests together. In order for the test to be launched in such a testsuite, it must be referred by one of the higher testset.init.

For example if you want to integrate your test inside the pulp_tests testsuite, you can open the testset.ini file in the pulp_tests folder. This file should look like:

```
[testset:pulp_tests]
files=  bench_tiny/testset.ini
                helloworld/testset.ini
```

You can just add the following line to this file to integrate your test:

```
hello/testset.ini
```

The line should give the relative path of your test testset.ini file, related to the testsuite testset.ini file.

To check that it is correctly integrate, you can dump the list of available tests by executing this command from the pulp_tests directory:

```
$ plptest tests
```

Your test will also be seen from higher testsuite, as for example from the SDK top testsuite.

Use the bench library

The bench library can be used to ease error and benchmark reporting. The idea is to declare a set of kernels which will be all called by the library one by one and a final report will be printed with the number of errors and cycles for each kernel, which is quite convenient when doing unit testing.

To get a test ready for this lib, add the following option to the previous command:

```
$ pulp-quickstart --path=test --test
```

run_testsuite must be called from the main entry point in order to start running the kernels. The kernels must be declared in the following structure which must be pass as an argument when calling *run_testsuite*:

```
static testcase_t testcases[] = {
  { .name = "test0",          .test = test0          },
  { .name = "test1",          .test = test1          },
  {0, 0}
};
```

Each kernel must have a precise prototype and are supposed to report the number of errors in the structure they receive as an argument and also to call the start and stop functions passed as arguments in order to specify the exact portion of code that must be benchmarked:

```
static void test0(testresult_t *result, void (*start)(), void (*stop)()) {
  volatile int i;
  start();
  for (i=0; i<1000; i++);
  stop();
  result->errors++;
}
```

Once run, it should print this kind of report at the end of the execution:

```
== test: test0 -> fail, nr. of errors: 1, execution time: 9079
== test: test1 -> fail, nr. of errors: 10, execution time: 938
==== SUMMARY: FAIL
NOT OK!!!!!
```

CHAPTER TWELVE

HOW TO USE THE FPGA

As for the RTL, there are 2 ways of using the FPGA

Let the SDK flow manage the bitstreams

The bistream for each supported chip is stored as a binary package in the artifactory server. When the SDK dependencies are retrieved, the correct bitstream is downloaded according to the configuration.

First the SDK must be properly initialized and configured for the FPGA (see *How to setup the SDK*).

Then you must download the bitstream for your architecture with this command:

```
$ plpbuild --p sdk deps
```

Now go to your application and execute this command to upload the bitstream to the FPGA:

```
$ make bistream
```

After this step you can use the FPGA as any other platform with the usual commands:

```
$ make clean all run
```

Provide an external bitstream

In case the bitstream for the chip you are using is not available or you want to synthesize it yourself, you can compile and run your application the usual way, just adding the following option:

```
$ make clean all run PLT_OPT=--emu-mapping=miaemu.bit.bin
```

Tips

When you launch a make command for the FPGA, you can specify the IP address of the FPGA with this option:

```
$ make clean all run PLT_OPT=--emu-addr=192.168.0.10
```

CHAPTER

THIRTEEN

HOW TO GET THE SDK

There are 2 ways of getting the SDK, either from sources or as a prebuilt binary package.

Getting a prebuilt binary package

Major SDK releases are deployed on the artifactory server in order to quickly get a ready version of the SDK. These versions are deployed at least every 4 weeks, but also for specific milestones.

Before accessing the server, make sure artifactory access is setup using this doc: *What must be configured before using or compiling the SDK*

Then the binary package can be retrieved by 2 different ways.

Using the downloader

Each deployed release has also an associated downloader which can be downloaded on the release section of the website here: https://iis.ee.ethz.ch/~haugoug/pulp/#releases

This way of getting the SDK is convenient for quickly getting a specific SDK.

This downloader is able to get the SDK as well as its dependencies, like the toolchains.

To download the SDK, once you got the downloader, execute the following command:

$ ./get-sdk-<sdk version> –path=<SDK root path>

The SDK root path should be the directory that will contain the SDKs and their dependencies. You can reuse the same directory for several SDKs in order to share the same dependencies for several SDKs.

Once everything has been downloaded you can see all the packages under the pkg folder. There is one directory per package and also one sub-directory per package version so that you can download and keep working on several versions without any conflict.

Using the top module

This way is more convenient when you plan to work on several SDKs at the same time or when you know you will partially rebuilt it as it makes it easy to start from a binary package, download the corresponding sources and rebuild it.

You must first checked-out the top module source code with this command and initialize the environment:

```
$ git clone git@iis-git.ee.ethz.ch:pulp-sw/pulp_pipeline.git
$ cd pulp_pipeline
$ git checkout <SDK tag>
$ source init.sh
```

Note that to make sure you get exactly the top module version that was used to produce an SDK, you must execute the appropriate 'git checkout' command. The given tag is the one of the SDK which can be either a git tag or a git commit hashtag.

Then you can use the plpbuild script to get the SDK and its dependencies:

```
$ plpbuild --p sdk get
```

Building from sources

You must first check out the top module source code with this command and initialize the environment:

```
$ git clone git@iis-git.ee.ethz.ch:pulp-sw/pulp_pipeline.git
$ cd pulp_pipeline
$ git checkout <SDK tag>
$ source init.sh
```

You have to configure it for the target you want to support:

```
$ plpconf --pulpChip=pulp4
$ source setup.sh
```

Then you have to execute this script to get SDK dependencies:

```
$ plpbuild --p sdk deps
```

Then in case you want to build the minimal part of the SDK needed to build and compile applications on all platforms, you have to execute:

```
$ plpbuild --g runtime --g platform checkout build
```

Tips

When building the SDK, it is by default build for all configurations specified in the top modules.ini file. In order to build it for a specific configuration, you can first generate a configuration file with the following script:

```
$ plpconf --pulpChip=pulp4
```

And then source the generated file:

```
$ source setup.sh
```

Then the SDK will be built only for this configuration.

In case you want to build the full SDK, you can execute:

```
$ plpbuld --p sdk checkout build
```

You can also skip the target configuration through plpconf in order to compile the SDK for all possible targets.

CHAPTER

FOURTEEN

HOW TO SETUP THE SDK

Environment setup

Whatever the way the SDK was retrieved (through downloader or from top module) or produced (from source), a setup file has been generated in the env folder that can be sourced to setup the terminal to be used with the SDK.

The file to source should look like *env/env-sdk-<version>.sh*. Just source it and you can then compile and run an application

Configuration setup

If you don't configure anything else, the applications will be compiled and launched for the default configuration. In case you want to specify a different configuration, you can define the following environment variable with the desired configuration:

```
$ export PULP_CONFIGS=pulpChip=mia:platform=rtl
```

The next time you launch an application, it will be compiled for Mia Wallace and run on the RTL platform.

Tips

A tool will soon be provided in order to know what are the possible configurations. Here are a few interesting ones:

- pulpChip: mia, pulp4, GAP, fulmine, pulp3 ...
- pulpCoreArchi: or1k, or10n, or10nv2, riscv, ri5cyv1, ri5cyv2, ri5cyv2-rvc ...
- pulpCompiler: gcc, llvm
- platform: gvsoc, fpga, rtl

CHAPTER

FIFTEEN

HOW TO DELEGATE JOBS TO JENKINS

Any step for compiling, testing or administrating the SDK can be done from the terminal in a classic way by executing stand-alone commands in the terminal.

However, it is sometimes more convenient to delegate some commands to Jenkins, for example if the command is long.

Jenkins is a continuous server that is always up and running. Some commands can be submitted to it so that they are dispatched to available worker workstations.

The script *plpjob* is provided in order to manage Jenkins jobs.

Submit a job

The minimal information to give for submitting a job is the command to be executed by Jenkins:

```
$ plpjob submit --cmd="echo Hello"
```

Be carefull, the command will be executed in a fresh environment by a Jenkins slave so the command must contain all what is needed to initialize the environment.

Several commands can be specified, and they will all be executed in sequence:

```
$ plpjob submit --cmd="export DUMMY_VAR=hello" --cmd="echo $DUMMY_VAR"
```

This is particularly useful when the environment must be configured as the commands are executed in sequence in the same shell, keeping the same environment between commands.

The command is by default launched on the same distribution than the one from which the job is launched. In order to launch it on a different distribution, the following option can be added:

```
$ plpjob submit --cmd="echo Hello" --distrib=CentOS_6
```

Once the job has been submitted, several information are displayed:

- The job name where the commands have been pushed
- The build number (this number is valid in the job)
- The URL where the enqueued job can be accessed from a Web browser

This information can be used to check the job status, as the submit command is by default asynchronous.

By default the job used is the job *package* but another job can be used by adding the option *–job=<job name>*, in order for example to gather commands of the same kind inside the same job (e.g. tests on a specific target). In this case, the job must have been created in Jenkins server by duplicating the *package* job.

The job can also be submitted and the command blocked until the job is finished by adding the *–block* option. In this case the return value of the submit command is the commands return value.

Check job status

Once a job has been enqueued, it is possible to check its status with the following command:

```
$ plpjob status --build=<job build number>
```

The build number is the one that has been displayed right after the job has been enqueued. If the the job has been enqueued to a specific Jenkins job (using option *–job*), the same Jenkins job must be specified.

If the job is not finished yet, the reported status will be *RUNNING*, otherwise it can be *SUCCESS* or *FAILURE*

The return value of this command is the one of the executed commands.

Get job output

As for checking the job status, the job output can be retrieved with a similar command:

```
$ plpjob output --build=<job build number>
```

The same remarks as for job status applies. It is also possible to get the job output while it is still running.

Wait for a job completion

It is possible to wait for a job completion using the following command:

```
$ plpjob wait --build=<job build number>
```

CHAPTER

SIXTEEN

HOW IS WORKING THE SDK CONTINUOUS INTEGRATION

Although the SDK is regularly released, there is a continous integration flow set-up in order to identify as soon as possible the possible reasons of broken features.

The entry point for seeing the results of this flow is here: https://iis-jenkins.ee.ethz.ch:8443/view/sdk

SDK versionning

The SDK is a set of modules with specific versions. The versions are specified in the top SDK module called pulp_pipeline. In order for an SDK to use more recent modules versions, this top-level module must be modified, which allows controlling what is exactly used by a specific SDK.

There are 2 important branches for this top-level module:

- **master**: this contains only stable versions of this SDK and no development should be committed directly to this branch.
- **integration**: this contains SDK versions which are good candidates for being integrated in the master branch. They are committed there in order to first validate them before merge to the master.

Per-commit builds

Every commit to the integration branch will trigger an SDK build and a deployment of the package to the artifactory server.

It will also trigger a quick non-regression testsuite (mainly on virtual platform, and a few quick tests on other platforms) so that a feedback is quickly available (after less than 1 hour).

These per-commit builds are not enough to say that the version can be integrated into the master branch, they should not be used to go-on with the developments.

These builds are using the gitlab CI feature to trigger builds on Jenkins. A file called .gitlab-ci.yml stored in pulp_pipeline root folder describes what must be launched and is interpreted by gitlab each time there is a commit.

Due to gitlab limitations, any commit in any branch of pulp-pipeline will actually trigger a build.

Nightly builds

In order to also check the impact of modifications in any of the SDK modules as soon as it is committed, there is also one build per night which is done of the latest modules versions.

For that a new version of pulp_pipeline is committed once per night in the integration branch, pointing to the latest modules versions.

This is done with a periodic Jenkins job that is using the plpadmin tool to checkout the latest modules and commit a new pulp_pipeline version. The Jenkins job can be accessed here: https://iis-jenkins.ee.ethz.ch:8443/view/sdk/job/sdk_nightly_build

Then the flow described above (per-commit build) is used with the difference that a full non-regression test suite is run, so that this version is ready to be merged in the master branch if it is stable enough.

Specific builds

When a specific version needs to be built and validated, the plpadmin tool can be used by hands to push a new pulp_pipeline version and trigger the automatic build.

CHAPTER SEVENTEEN

HOW TO BUILD SDK COMPONENTS

The SDK contains several components (virtual platform, OMP runtime, OVX runtime) that can be compiled separatly insted of compiling the whole SDK.

OpenVX runtime and libs

The OpenVX Pulp implementation is currently available only on a prototyping architecture derived from pulp-dev and called pulpevo. Thus to have OpenVX support in the SDK, it must be compiled with the following commands:

```
$ git clone git@iis-git.ee.ethz.ch:pulp-sw/pulp_pipeline.git
$ cd pulp_pipeline
$ source init.sh
$ plpconf --pulpChip="pulpevo" --pulpRtVersion=bench --pulpOmpVersion=ovx
$ source setup.sh
$ plpbuild --p sdk deps checkout build
```

Note that when compiling the whole SDK with the default command, the OpenVX support is also added.

In order to not build everything but just the OpenVX part (once the SDK has been compiled once to get all the tools), you can replace the plpbuild command by this one:

```
$ plpbuild --g runtime_ovx --g ovx checkout build
```

The group *runtime_ovx* will compile all the low-layer runtime while the group *ovx* will compile the OpenVX library and the kernels.

CHAPTER

EIGHTEEN

HOW TO PROFILE AN OPENMP APPLICATION

Callgraph profiling

On RTL simulator and virtual platform, a PC-based callgraph for a single core can be generated with the following command:

```
make clean all run KCG=0
```

This will generate the callgraph for the specified core number, 0 in this case.

The callgraph can be seen with KCachegrind tool with (the actual path to the kcg file is displayed at the end of the simulation and may be different):

```
kcachegrind build/kcg.txt
```

Kcachegrind is an open-source tool whose documentation can be found here: http://kcachegrind.sourceforge.net/html/Home.html

Here are a few hints:

- On the top-right corner, you'll find the event currently being watched. The platform is registering the events from the core HW counters and thus any HW event can be viewed. The first one (*PC*) is just based on the PC information and is equivalent to *Instructions*. All the others are HW events, you can refer to the core datasheet for more information. The most common event is *Cycles* as it will show the cycles spent.

And the most interesting panels:

- **Flat Profile**. It is by default the left panel. This view will show the time spent in each function, both including everything or just including the time really spent in the function not the one spent in the called functions
- **Source Code**. This shows the time spent on each source code line, which is nice for identifying the hot spots. This will also show the functions called, and how many time.
- **Call graph**. This shows how the time spent in a function is spent between the functions called by this one.

Execution traces profiling

With OMP on the virtual platform, more information about the execution context can be obtained with:

```
make clean all run pulpRtVersion=profile0
```

This activates a version of the runtime instrumented for generating context information through HW traces.

They can be visualized either with Android traceview tool (only on Ubuntu) with this command:

```
traceview build/tvDump
```

Or as a textual callgraph:

```
cat build/cg_cluster0_core0.txt
```

The actual paths to the files might also be different and are displayed at the end of the simulation.

Kernel statistics

The main issue of the execution traces is that it does not give any application context, and thus it is hard to link the results with the application code.

To overcome this, it is possible to instrumentate the application to give the missing context, by classifying the source into kernels.

To use this support, the following file must be included:

```
#include "hwTrace.h"
```

The file is located here in the SDK: *install/include/hwTrace.h*. More information about this support can be found in this file.

An example can be found in apps/gomp_tests/kernelTrace.

Here are the main things to know:

- In order to better identify the performances of an application, this support allows breaking the whole code into kernels and sees the performances of each kernel.
- Kernels must first be declared with the following call:

  ```
  pulp_trace_kernel_declare(0, "kernel 0");
  ```

  The first argument is the kernel identifier which must be unique and which will be used later on to associate traces to this kernel. The second argument is a string that will be used by the profiling tools to report results for this kernel.
- The execution of a kernel can then be caught with the following sequence:

  ```
  pulp_trace_kernel_start(0, 1);
  // Put the kernel code here
  pulp_trace_kernel_stop(0, 1);
  ```

  All instructions executed between these 2 calls will be accounted on this kernel. The first argument to both calls is the kernel identifier specified when declaring the kernel. The second argument should be 0 if we just want the report about the instructions or 1 if we also want the report for all core HW counters (which is more intrusive). If everything goes well, you should see something like this at the end of the simulation:

  ```
  +----+----------+----+------------+------------+------------+--------------+--------------+-----
  | ID |     name | nb | avg cycles | min cycles | max cycles | total cycles | Cycles       | Instructio
  +----+----------+----+------------+------------+------------+--------------+--------------+-----
  |  0 | kernel 0 | 10 |       1532 |       1365 |       2680 |        15329 |        13233 |
  +----+----------+----+------------+------------+------------+--------------+--------------+-----
  ```

- The kernel instructions can also be seen using traceview by opening the traceview report (see above). The kernel execution will now be displayed as a specific event which allows better understanding the global execution.
- If the kernel is executed several times, and thus the execution goes several times through the sequence start -> stop, each execution will be registered and the number of executions will be reported (column *nb*). In this case,

the column *avg cycles* is an average over all these executions, *min cycles* and *max cycles* are the mininal and maximal cycles count seen over the executions. All the other statistics are cumulated over the executions.

- In case some instructions executed inside a start -> stop sequence should not be accounted on the kernel (for example to not include some debug code), it is possible to pause and resume the profiling with these calls:

```
// Kernel code
pulp_trace_kernel_pause(0, 1);
// This code here won't be accounted on the kernel
pulp_trace_kernel_resume(0, 1);
// Kernel code
```

The arguments are the same as for start and stop

- The number under *Cycles* is the number of cycles where the core is active, while *total cycles* is also including the number of cycles where the core is sleeping.

CHAPTER NINETEEN

HOW TO VALIDATE AN SDK RELEASE

Automatic non-regression tests

The usual automatic tests should be run and have a good coverage

Manual tests

The following manual tests should be run as they cannot be automated:

- **Callgraph profile**. Go to *apps/gomp_tests/bench* and run:

```
$ make clean all run KCG=0
```

  Then open the kcachegrind report and check it looks good. Also check it on RTL platform.

- **Traceview report**. Got to *apps/gomp_tests/kernelTrace* and run:

```
$ make clean all run pulpRtVersion=profile0
```

  Then open the traceview report and check it looks good. Some OMP events should be seen on master and slave cores, and also 4 kernel executions. Check that it is coherent with the textual version of the call graph (e.g. cg_cluster0_core0.txt) and also the textual kernel statistics report.

CHAPTER TWENTY

HOW TO RELEASE AN SDK

There are 3 steps to build, fully validate and deploy an SDK release:

- Validate a precise version of the SDK
- Tag it, and validate again with the new tag
- Deploy the SDK

Validate a precise version of the SDK

To be sure that nothing has been left aside, you must checkout, build and validate a precise SDK version from scratch. Otherwise, it could happen that one package has not been deployed, or one commit has not been pushed. Also if the SDK is not rebuilt from scratch, it could work thanks to an old file which is still there in the SDK and it would make it impossible to rebuild the same SDK later on.

An SDK tag corresponds to the SDK that you would get by checking out and building the SDK from the top SDK module with the version that you are tagging. The SDK version that you are tagging is the git version of the SDK top module as this version encures you that you can always get exactly the same SDK from this version.

First checkout the following repository containing all the tools for releasing an SDK:

```
$ git clone git@iis-git.ee.ethz.ch:pulp-sw/pulp_sdk_release.git
$ cd pulp_sdk_release
```

Then execute the following script:

```
$ ./plpadmin --version=<sdkGitVersion> --tag=<sdkTag> checkout build test --submit
```

sdkGitVersion is the version of the top SDK module that you want to tag. sdkTag is the SDK tag that will be produced.

This script will first checkout the sources and get the dependencies, will then build the SDK, launch the testsuite and report the results.

Due to the *–submit* option, all these steps will be done on Jenkins, the option can be removed in order to execute them locally. As the SDK is released for several Linux distributions, this may trigger several SDK builds.

If you are happy with the results you can go on with the process, otherwise the issues has to fixed and the process start over with a new SDK version.

Tag the SDK and validate it again

Execute this command to tag the SDK:

```
$ ./plpadmin --version=<sdkGitVersion> --tag=<sdkTag> tag
```

This will tag it locally and push the tag to the server. Note that it needs to commit that contains the right tag for the SDK version in versions.ini. This commit is pushed by default to the release branch can be changed with option *–branch*.

In order to be sure the tag has been properly pushed, validate again the SDK, using the tag as the SDK version:

```
$ ./plpadmin --version=<sdkTag> checkout build test --submit
```

If you are happy with the results you can go on and deploy the SDK

Deploy the SDK

In case the SDK was built locallu, first deploy it to the artifactory:

```
$ ./plpadmin --version=<sdkTag> deploy
```

And deploy the documentation:

```
$ ./plpadmin --version=<sdkTag> deploydoc
```

Now you can get the deployed SDK through the downloader and validate it again.

Simplified procedure using jenkins

All these steps can be can done in a simpler way using the Jenkins server, following these steps:

- Wait until one SDK version from the integration branch is stable enough
- Merge it to the master branch
- Launch the job sdk_tag (https://iis-jenkins.ee.ethz.ch:8443/job/sdk_tag) and put the tag name
- The documentation is not yet deployed from Jenkins so you have to execute this command: ./plpadmin –version=<tag name> checkout deploydoc

Current SDK tagging flow with Jenkins

Pre-release tags

Before a final tag is made, intermediate tags are done. They are done from the master branch and have a tag name like 2016.3-b5. The procedure above can be used for these tags. These tags reasonably stable and should only be used by people who wants to closely follow SDK evolutions.

Final tags

A final tag is a stable tag meant to be used by everyone for at least a few weeks. It is released the same way than pre-release tags

Patch tags

These tags are done on a final tag to bring bug fixes. Contrary to the other tags they are done from the tag branch instead of the master. Bug fixes can indeed be committed and tested on the tag branch, and when it is stable enough a new tag is made from there.

CHAPTER

TWENTYONE

HOW TO CUSTOMIZE MAKEFILE FLAGS

When the runtime or an application is compiled, defaults flags are used that can be either extended or replaced (at least part of them).

How to customize flags

Flags can be modified by defining the proper makefile variable (either on the command line or in the makefile) like in the following:

```
$ make clean all run PULP_CFLAGS=-Werror
```

They can also be customized by defining the proper environment variable in order to make the customization more permanent like in the following:

```
$ export PULP_CFLAGS=-Werror
```

Which flags can be customized

Additional C flags can be specified using the PULP_CFLAGS variable and LD flags with PULP_LDFLAGS. The specified flags are happened to the default ones.

Default compiler architecture flags can also be replaced by defining the PULP_ARCH_CFLAGS variable like in the following:

```
$ make clean all PULP_ARCH_CFLAGS=-march=IMXpulpv2
```

PULP_ARCH_LDFLAGS and PULP_ARCH_OBJFLAGS can be defined for architecture flags for linker and obj-dump

Runtime flags customization

As runtime modules are compiled through a script, the only way to customize the flags used for compiling runtime is to define an environment variable, it is not possible to do it through make variables.

For example in order to recompile the full runtime for a specific core, the following commands can be executed:

```
$ export PULP_ARCH_CFLAGS=-march=IMXpulpv2
$ plpbuild --g runtime clean build
```

Application flags customization

Application flags can be customized either through environment variables or make variables. However it is better to do it through environment variables to make sure runtime and application compilations are well aligned.

CHAPTER

TWENTYTWO

HOW TO MANAGE GITLAB FROM THE SHELL

A few gitlab features can be accessed from the shell using the script plpgitlab.

Creating a user

This should still be done using the web interface as the API to do it from the shell is a bit limited

Getting user information

This can be done using the following command:

```
$ plpgitlab userinfo --user=<name>
```

Any user that match the given string in any of its properties will be displayed, so there could be severall. To restrict to a single user, the following option can be used instead to give the ID of the user (the one displayed in the information):

```
$ plpgitlab userinfo --user-id=<id>
```

The access rights for a user can also be displayed by adding the following option to specify a group of gitlab groups for which they must be displayed:

```
$ plpgitlab userinfo --user-id=<id> --group=<group>
```

There is currently 3 available groups: sdk, rtl and greenwave. This should then display each gitlab group found and the associated access rights.

Modify access rights

The access rights for a group can be modified using the following command:

```
$ plpgitlab memberadd --user-id=136 --group=sdk --access=reporter
```

This will either update the access rights or add them to the specified user for the gitlab groups found.

CHAPTER

TWENTYTHREE

HOW TO TAG THE WHOLE STUFF

Specifications tag

The first thing to do is to tag the specifications. The name depends on the chip:

- GAP is currently tagged this way: gap-<day>-<month>-<year>. If several tags must be done the same day, the day is like this <day>_<number>

The specifications must be pushed to the corresponding git repository and tagged in it using the tag name described above.

For GAP the repository is git@iis-git.ee.ethz.ch:pulp-project/GAP-specifications.git

RTL tag

Once the specs are tagged, the RTL can be tagged several tags with the same specs to reflect the various implementations of the same specs.

The convention for the tag name is the following <specs tag>-rtl-<build number>.

For example if the specs tag is gap-07-09-2016, the first RTL tag for it will be gap-07-09-2016-rtl-0.

The last number can be incremented everytime there is a new RTL tag of the same specs.

SDK tag

The convention is the same as for RTL except that it is sdk instead of rtl. Thus for the example above the first tag name would be gap-07-09-2016-sdk-0.

HOW TO DUMP UDMA CHANNELS

This is a feature available only on the virtual platform in order to dump into a file all the data that goes through a specific channel.

How to activate it

This feature can be controlled through the **–periph-dump** option like in this example:

```
make run PLT_OPT="--periph-dump=spim0Tx:$PWD/spim0Tx.txt"
```

This option takes a value which specifies first the channel name to be dumped, and second the file path where it should be dumped, separated by a colon.

Several options can be specified in order to dump several channels:

```
make run PLT_OPT="--periph-dump=spim0Tx:$PWD/spim0Tx.txt --periph-dump=spim0Rx:$PWD/spim0Rx.txt"
```

How to get the channel name

The channel name depends on the target architecture. The best way to get those names is to first dump UDMA traces, and to identify the interesting channels in the traces:

```
make run PLT_OPT="--gv-trace-file=stdout --gv-trace=udma"
```

Specific peripherals

A few peripherals provide specific options in order to get the data that goes through them, in order to get better formatted data.

Here is an example for the UART which disables the dump of transmitted data to the standard output and instead redirect it to a file:

```
make run PLT_OPT="--uartFile=$PWD/uart.txt"
```