Session 4: module integration with a pipelined CPU

The objective of this session is to document the integration of the developed module(s) into the pipelined CPU of choice. We have chosen to organise the documentation process of integrating the module's as follows.

CPU verification

Available options for verifying that the Ariane CVA6 processor is working correctly and why it is relevant for our case.

Verilator and CVA6

How to obtain waveform files for user-space applications, i.e. simple, gcc-compiled C files in our case.

Theoretical integration and hurdles

ECC module integration into a theoretical cache structure and the reality of how the Ariane CVA6 processor gets in the way of this.

Indice

1. CPU verification **2**

2. Verilator, CVA6 and waveforms **11**

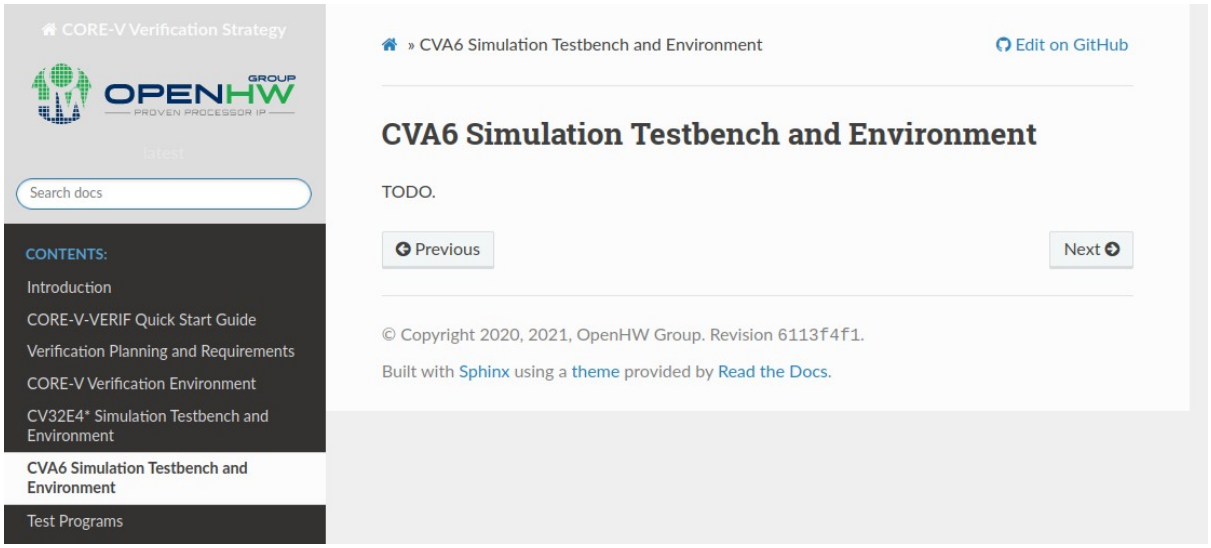
3. Theoretical integration and hurdles **24**

1. CPU verification

Introduction to the “core-v-verif” repository

OpenHW Group, the folks behind the development and maintenance of the Ariane CVA6 processor (among others), keep in separate repositories the actual RTL of CVA6 and the scripts and utilities for its verification. Although CVA6’s source code repository has some instructive set-up steps to verify that one has correctly installed the RISC-V toolchain or the proxy kernel or the spike disassembler, serious modifications or extensions to the processor simply cannot be asserted and verified by running a simple hello world C program. Thus, the “core-v-verif” repository.

The “core-v-verif” repository allows for verification of some of OpenHW Group’s RISC-V cores. There are seven cores in total under their supervision, two “application-class” cores in CVA{5,6} (with M, S, U privilege levels for operating system support and the like) and five smaller embedded cores in CV32E40{P,X,S}, CV32E41P and CVE2. Only a subset of those seven can be verified by the “core-v-verif” repository and, in the case of CVA6, only a certain version (a specific commit) is “supported“ by this repository. Even if support is given for a certain core, as it is the case for the CVA6, the documentation for it can appear way too scrambled and disjointed through the repository’s READMEs or the readthedocs page to make sense of it, not to mention… Well, see picture below.



To be fair, the readthedocs page of the “core-v-verif” repository does specify that the CV32E40P is the more mature of the projects and, thus, it is a good place to start since *“the core testbench is very simple and runs with open-source tools”*. The verification “tutorial” uses this core as an example, but the maturity of the project does not excuse the fact that the CV32E40P core has a completely different directory structure to CVA6’s and, therefore, one cannot follow the same verification steps for CVA6 as for CV32E40P (the one “officially” supported).

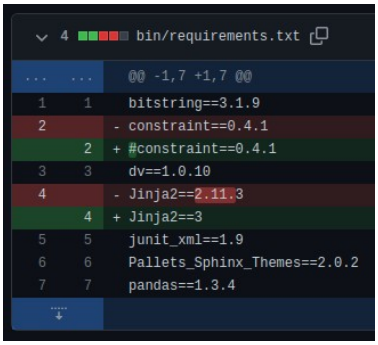
Regardless, the purpose of this repository is clear: provide a comprehensive set of testbenches with which to establish the status of a certain (modified) core with minimal external-tool headaches. The verification of a core is, to our understanding, carried out by running instruction-specific testbenches whose spike output is considered as the baseline functioning of what the core should be doing. According to the similarity between spike's results and the core's output, a testbench is either passed or failed. This would be the ideal framework to perform any extensions to the base RTL of some OpenHW Group processor.

Making things run

For starters let's see how to run the CV32E40P core's helloworld tutorial. Looking at the first steps, we already satisfy most of the requirements:

> 1. A Linux machine (CORE-V-VERIF has been successfully run under Ubuntu, Debian and CentOS).
> 2. Python3 and a set of plug-ins. The current plug-in list is kept in *$CORE_V_VERIF/bin /requirements.txt*.
> 3. A GCC cross-compiler (aka "the Toolchain"). Even if you already have a RISC-V toolchain, please do follow that link and read **TOOLCHAIN.md** for recommended ENV variables to point to it.
> 4. Verilator.

1. We established in Lab 1's report that Ubuntu 20.04 was a working distribution for CVA6's repository specifics.
2. The Python requirements are fixed as of November.
    a. We found this pretty funny, since we encountered problems with the "constraint" package back in the day. We also took the route of completely ignoring it, as they do in the commit's fix, since it is a PyPi package that has not been touched since 2010 and caused some errors when trying to install it.



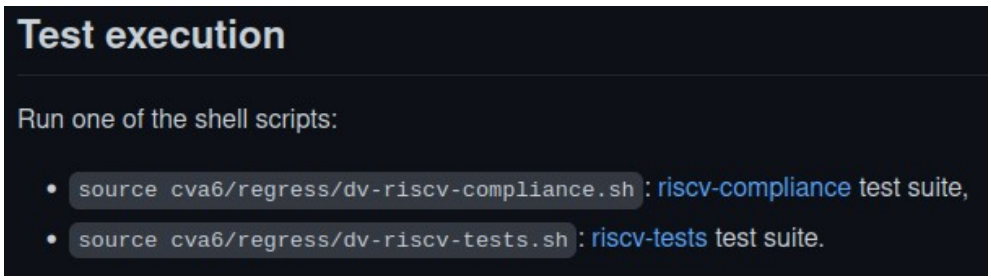
3. The RISC-V toolchain was already a must for making the CVA6 work, although this "core-v-verif" repository requires some more environment variables declared for it to work properly. The Ariane CVA6 processor is covered in full in the next section. This section on CVA6's verification only requires the dependencies specified back in Lab 1, but, regardless, a step-by-step installation script is described/included in the next CVA6-specific section of this document to get up to speed. Just add the following variables to .bashrc if you, the reader of this document, have already managed to make CVA6 go according to our instructions:

```
export CV_SW_TOOLCHAIN=/riscvstuff
export CV_SW_PREFIX=riscv64-unknown-elf-
```

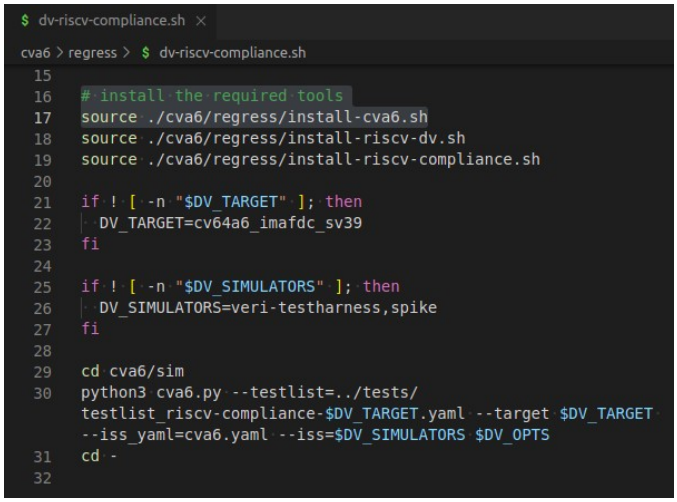
4. Verilator was also installed from CVA6's days, plus, the "core-v-verif" repository makes sure to build a local copy later down the line ("core-v-verif" downloading its own version of verilator causes some problems down the line in the CVA6's repository related to the `VERILATOR_ROOT` environment variable, but all in due time).

Installing the requirements and "making" the CV32E40P processor for running its helloworld program really is as simple as advertised in the readthedocs page. However, do not hold your breath waiting for the helloworld to finish, as it takes its sweet time.

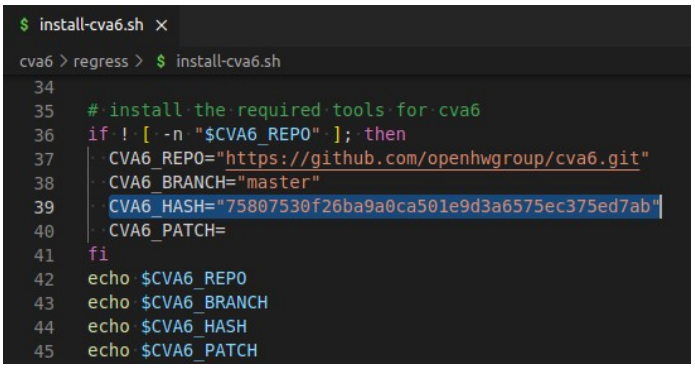
The CVA6 processor, however, does not adhere to this tutorial and is instead built and tested by executing two possible bash scripts. This information is available at CVA6's README in the "core-v-verif/cva6/" directory. Hopefully this will later be clarified in the documentation section for the CVA6, once it is finished and well past its current "TODO" state.



These two scripts, aside from executing some tests for the processor, will install spike and verilator into the "core-v-verif/tools/" directory of the repository, regardless of whether you have working versions of these tools already installed in your machine. This makes things simpler, as the installation of these tools from CVA6's repository was somewhat of a pain, but, upon close inspection, the installation scripts only change the installation directory of the tools while running the same installation scripts from the standalone CVA6 repository. This way, any automated verification-test scripts call the utilities under the "core-v-verif/tools/" directory instead, which is not that different from what we did for the CVA6 back in the day. Regardless, let's do a little deepdive into these scripts, since some little fixes are needed to get things working.



Let's look at, say, the "dv-riscv-compliance.sh" script. It does not matter which of the two bash scripts we go looking into right now since they both call the same files pertinent to the following fix. Line 17 of the script makes the call to the "install-cva6.sh" script, responsible for installing the local versions of spike and verilator, and whose structure is all too similar to CVA6's repository files (thankfully). This script, by the way, is also responsible for git-cloning the only verifiable version (apparently) of the CVA6 processor, specifically, commit number "758075" from May of 2022:



We will later see in the CVA6 section of this report that newer versions of the processor can be run and verified with no issue whatsoever or no failed testbenches, but everything in due time. What's important here right now is that there are going to be some Python-related errors popping up stemming from the call in the "install-cva6.sh" script to the "cva6.py" file. Running the "dv-riscv-compliance.sh" script for the first time should output the following error (we had already run the script previously, so it detects the local spike and verilator in the "core-v-verif/tools/" directory):

```
asier@asier-ThinkPad-L380:~/core-v-verif$ source cva6/regress/dv-riscv-compliance.sh
Using Verilator from cached directory.
https://github.com/openhwgroup/cva6.git
master
75807530f26ba9a0ca501e9d3a6575ec375ed7ab

Using Spike from cached directory.
https://github.com/google/riscv-dv.git
master
96c1ee6f371f2754c45b4831fcab95f6671689d9

https://github.com/riscv/riscv-compliance.git
main
220e78542da4510e40eac31e31fdd4e77cdae437
../../../cva6/regress/riscv-compliance.patch
Traceback (most recent call last):
  File "cva6.py", line 27, in <module>
    from dv.scripts.lib import *
ModuleNotFoundError: No module named 'dv.scripts'
/home/asier/core-v-verif
asier@asier-ThinkPad-L380:~/core-v-verif$
```

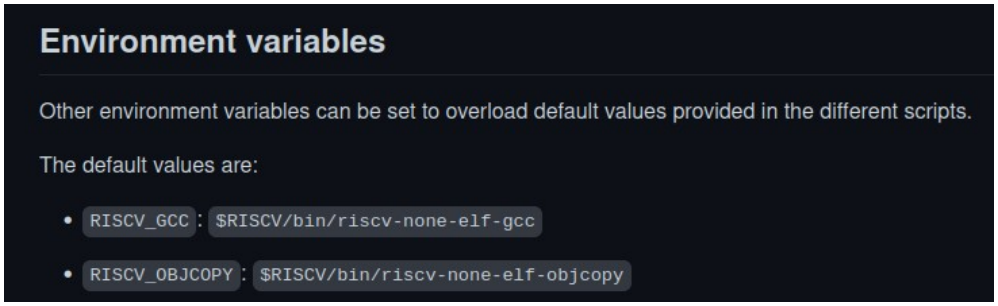
Inspecting the “cva6.py” file, the error happens to be related to broken Python imports. There is a “dv/” directory in the “cva6/sim/” folder of the verification repository from which the imports are trying to obtain some code. However, this “dv/” directory cannot be treated as a Python module from which to import stuff since it is lacking a “__init__.py” file within it. Creating an empty file of this name fixes the error.

```
asier@asier-ThinkPad-L380:~/core-v-verif$ source cva6/regress/dv-riscv-compliance.sh
Using Verilator from cached directory.
https://github.com/openhwgroup/cva6.git
master
75807530f26ba9a0ca501e9d3a6575ec375ed7ab

Using Spike from cached directory.
https://github.com/google/riscv-dv.git
master
96c1ee6f371f2754c45b4831fcab95f6671689d9

https://github.com/riscv/riscv-compliance.git
main
220e78542da4510e40eac31e31fdd4e77cdae437
../../../cva6/regress/riscv-compliance.patch
Traceback (most recent call last):
  File "cva6.py", line 27, in <module>
    from dv.scripts.lib import *
ModuleNotFoundError: No module named 'dv.scripts'
/home/asier/core-v-verif
asier@asier-ThinkPad-L380:~/core-v-verif$ touch cva6/sim/dv/__init__.py
asier@asier-ThinkPad-L380:~/core-v-verif$ source cva6/regress/dv-riscv-compliance.sh
Using Verilator from cached directory.
https://github.com/openhwgroup/cva6.git
master
75807530f26ba9a0ca501e9d3a6575ec375ed7ab

Using Spike from cached directory.
https://github.com/google/riscv-dv.git
master
96c1ee6f371f2754c45b4831fcab95f6671689d9

https://github.com/riscv/riscv-compliance.git
main
220e78542da4510e40eac31e31fdd4e77cdae437
../../../cva6/regress/riscv-compliance.patch
Mon, 19 Dec 2022 02:27:38 INFO     Creating output directory: /home/asier/core-v-verif/cva6/sim/out_2
022-12-19
Mon, 19 Dec 2022 02:27:38 INFO     Processing regression test list : ../tests/testlist_riscv-complian
ce-cv64a6_imafdc_sv39.yaml, test: all
Mon, 19 Dec 2022 02:27:38 INFO     Found matched tests: rv64im-REMUW, iterations:1
Mon, 19 Dec 2022 02:27:38 INFO     Found matched tests: rv64im-MULW, iterations:1
Mon, 19 Dec 2022 02:27:38 INFO     Found matched tests: rv64i-SRAW, iterations:1
Mon, 19 Dec 2022 02:27:38 INFO     Found matched tests: rv64i-ADDW, iterations:1
Mon, 19 Dec 2022 02:27:38 INFO     Found matched tests: rv64i-ADDIW, iterations:1
```

However, yet another thing breaks a bit later:

```
Traceback (most recent call last):
  File "cva6.py", line 1144, in <module>
    main()
  File "cva6.py", line 1081, in main
    run_assembly(path_asm_test, args.iss_yaml, args.isa, args.target, args.mabi, gcc_opts,
  File "cva6.py", line 423, in run_assembly
    run_cmd_output(cmd.split(), debug_cmd = debug_cmd)
  File "/home/asier/core-v-verif/cva6/sim/dv/scripts/lib.py", line 192, in run_cmd_output
    output = subprocess.check_output(cmd)
  File "/usr/lib/python3.8/subprocess.py", line 415, in check_output
    return run(*popenargs, stdout=PIPE, timeout=timeout, check=True,
  File "/usr/lib/python3.8/subprocess.py", line 493, in run
    with Popen(*popenargs, **kwargs) as process:
  File "/usr/lib/python3.8/subprocess.py", line 858, in __init__
    self._execute_child(args, executable, preexec_fn, close_fds,
  File "/usr/lib/python3.8/subprocess.py", line 1704, in _execute_child
    raise child_exception_type(errno_num, err_msg, err_filename)
FileNotFoundError: [Errno 2] No such file or directory: '/riscvstuff/bin/riscv-none-elf-gcc'
/home/asier/core-v-verif
asier@asier-ThinkPad-L380:~/core-v-verif$
```

This means that the verification repository's version of CVA6 is trying to use a "riscv-none-elf-gcc" compiler. However, provided that you, the reader, have already managed to run CVA6 according to our instructions, your "riscv-*-elf-gcc" compiler should be named "riscv64-unknown-elf-gcc". As such, and by noticing that CVA6's README in "core-v-verif" mentions this "none" compiler, we should add the following variables to our .bashrc (we also added a second additional variable since the script will also complain about the "-objcopy" thing if not defined properly with the "unknown" keyword):



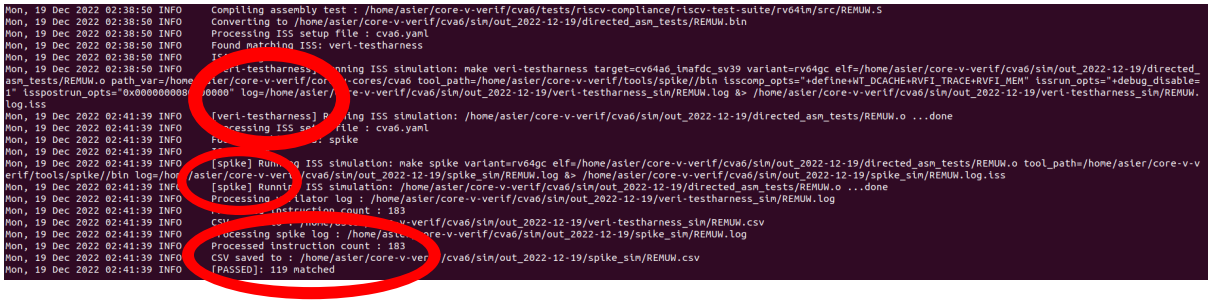

```
export RISCV_GCC=$RISCV/bin/riscv64-unknown-elf-gcc
export RISCV_OBJCOPY=$RISCV/bin/riscv64-unknown-elf-objcopy
```

Everything should be running now

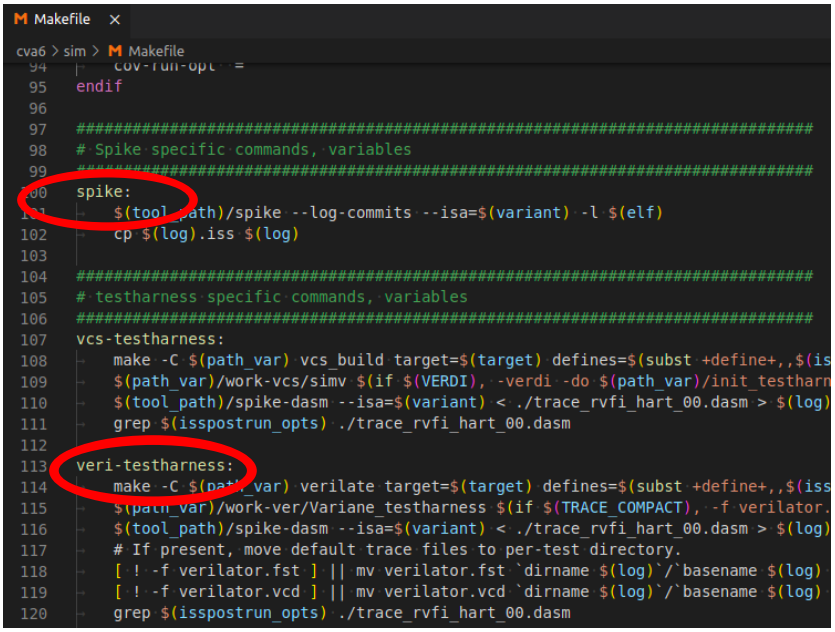
Executing the "dv-riscv-compliance.sh" script as "source cva6/regress/dv-riscv-compliance.sh" should yield something like this:

```
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32imc-C-MV, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32imc-C-LUI, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32imc-C-J, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32imc-C-ANDI, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32Zicsr-I-CSRRC-01, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32Zicsr-I-CSRRS-01, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32Zicsr-I-CSRRSI-01, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32Zicsr-I-CSRRW-01, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32Zicsr-I-CSRRCI-01, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32Zicsr-I-CSRRWI-01, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amoxor_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amoadd_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amoor_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amomin_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amoand_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amomaxu_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amoswap_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amominu_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ua-amomax_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fclass, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-ldst, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fmadd, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-recoding, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fcvt, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fcmp, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fcvt_w, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fadd, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fmin, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-move, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32uf-fdiv, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fclass, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-ldst, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fmadd, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-recoding, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fcvt, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fcmp, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fadd, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fmin, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     Found matched tests: rv32ud-fdiv, iterations:1
Mon, 19 Dec 2022 02:38:50 INFO     CVA6 Configuration is
Mon, 19 Dec 2022 02:38:50 INFO     Compiling assembly test : /home/asier/core-v-verif/cva6/tests/risc
v-compliance/riscv-test-suite/rv64im/src/REMUW.S
Mon, 19 Dec 2022 02:38:50 INFO     Converting to /home/asier/core-v-verif/cva6/sim/out_2022-12-19/dir
ected_asm_tests/REMUW.bin
Mon, 19 Dec 2022 02:38:50 INFO     Processing ISS setup file : cva6.yaml
Mon, 19 Dec 2022 02:38:50 INFO     Found matching ISS: veri-testharness
Mon, 19 Dec 2022 02:38:50 INFO     ISA rv64gc
Mon, 19 Dec 2022 02:38:50 INFO     [veri-testharness] Running ISS simulation: make veri-testharness t
arget=cv64a6_imafdc_sv39 variant=rv64gc elf=/home/asier/core-v-verif/cva6/sim/out_2022-12-19/directed
_asm_tests/REMUW.o path_var=/home/asier/core-v-verif/core-v-cores/cva6 tool_path=/home/asier/core-v-v
erif/tools/spike//bin isscomp_opts="+define+WT_DCACHE+RVFI_TRACE+RVFI_MEM" issrun_opts="+debug_disabl
e=1" isspostrun_opts="0x0000000080000000" log=/home/asier/core-v-verif/cva6/sim/out_2022-12-19/veri-t
estharness_sim/REMUW.log &> /home/asier/core-v-verif/cva6/sim/out_2022-12-19/veri-testharness_sim/REM
UW.log.iss
```

This means that the CVA6 processor will be running all the “Found matched tests” and will be done running all of them in an hour or so. Taking a closer look at the console output, we can begin to understand what the verification process actually entails:

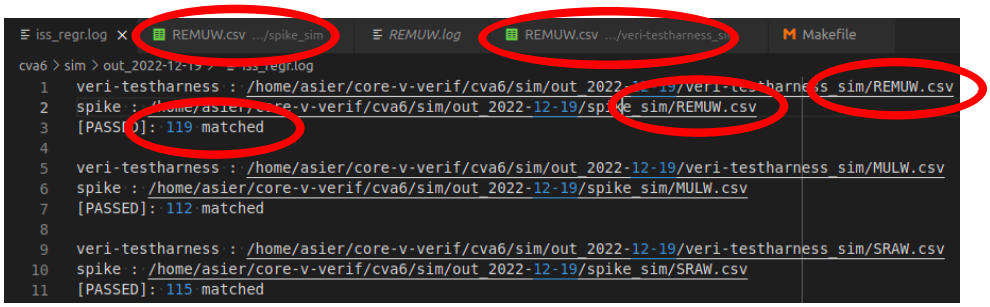


There’s three parts to the above console output: the “verilated” CVA6 model running the test, the spike disassembler running the same test and the similarity check of both test runs. Anything that is preceded by [veri-testharness] is synonymous with the verilated CVA6 model, as the Makefile in “cva6/sim/” gives away:

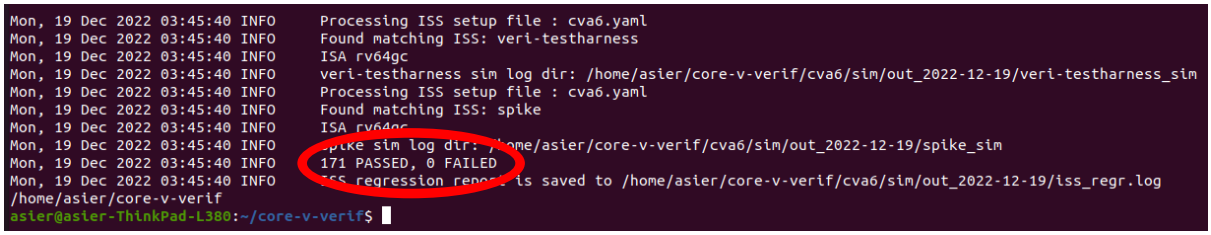


Similarly, anything preceded by [spike] is synonymous with the spike disassembler running the tests. The third part of the execution of a single test has to do with (we assume) checking whether the output of spike and the output of the verilated processor model match in a sufficient number of instructions for it to be considered correct or incorrect. To our understanding, the ideal output of spike and its executed instructions is what a processor model should be executing, while the verilated model's output is checked for similarity with spike's.

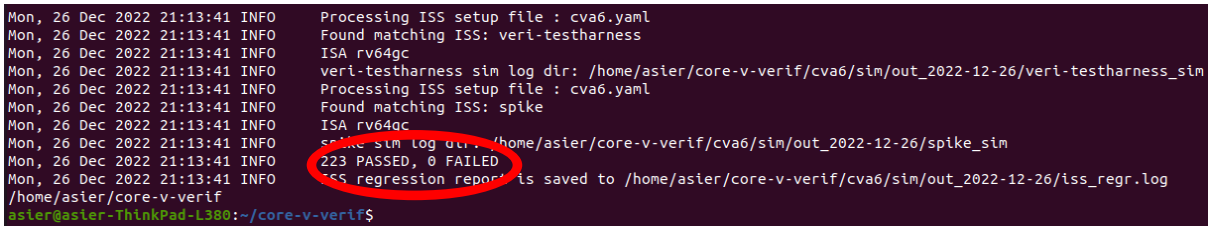
In the case of the REMUW testbench above, we couldn't figure out what the "119" number was referring to (**all the testbench results will be uploaded to the project's repository, along with the testbench runs that generated .fst waveform files as well!**). Inside the "cva6/sim/out_*/" directory are .log files and .csv files for "spike_sim/" and "veri-testharness_sim/" directories, containing information on the execution of each testbench. Both REMUW.csv files in the "spike_sim/" and "veri-testharness_sim/" directories are identical to each other. The log files are somewhat of a dead end too, not much human-readable information can be found, although the spike files do have some logging messages such as ">>>> _start" or ">>>> begin_testcode". An abbreviated version of all testbench results can be found in the "cva6/sim/out_*/iss_regr.log" file (see figure below).



The "iss_regr.log" file should also have, at the end, the following figure's console output, where it is stated that 171 total testbenches were passed and none were failed:



The execution of the other RISC-V test script “dv-riscv-tests.sh” should output 223 passed tests, also with no failed tests:



With this, we have established how to run the verification workflow of the CVA6 Ariane processor with the tools and methodologies provided by the “core-v-verif” repository (albeit with some minor fixes along the way). Now, in the next section, we’ll explore how to obtain waveform files from these verification testbenches so as to get a complete understanding of how the processor behaves under any given testbench and (provided that a new extension/module has been added to the processor and must thus be verified too) to see how the new extension/module to the processor acts, either correctly or not.

2. Verilator, CVA6 and waveforms

The previous section managed to get the verification of the base processor working, which is fine and necessary to confirm that a new processor module isn't altering its behaviour in some unexpected, undesired manner. A missing aspect of the previous verification process, however, are the waveforms required to assert, in turn, that the new processor module isn't sitting idly while it sees processor verification testbenches fly by, unresponsive to them. It is not sufficient to confirm that all tests have passed successfully, we must see that the new processor module is behaving as expected by looking at waveform signals. For this, we must obtain a waveform-generating version of the CVA6 processor. In fact, CVA6's repository specifies how to do so in its README page:

Build Model and Run Simulations

Build the Verilator model of CVA6 by using the Makefile:

```
make verilate
```

To build the verilator model with support for vcd files run

```
make verilate DEBUG=1
```

Let's take a look at how to go about this and what the DEBUG flag represents. As an aside, the last point of this section includes a record of all the necessary steps to download the CVA6 repository, so everything and anything should be runnable after this section to replicate what we claim we have worked on in this report.

Verilator and waveforms

The verilated model of the processor, as the name suggests, uses verilator to compile all the RTL sources into an binary-like executable to run tests and other elf binaries on. Verilator's documentation pages have the following information on waveform files:

From https://www.mankier.com/1/verilator:

```
--top <topname>             Alias of --top-module
--top-module <topname>      Name of top level input module
--trace                     Enable waveform creation
--trace-coverage            Enable tracing of coverage
--trace-depth <levels>      Depth of tracing
--trace-fst                 Enable FST waveform creation
--trace-max-array <depth>   Maximum bit width for tracing
--trace-max-width <width>   Maximum array depth for tracing
--trace-params              Enable tracing of parameters
--trace-structs             Enable tracing structure names
--trace-threads <threads>   Enable FST waveform creation on separate threads
--trace-underscore          Enable tracing of _signals
 -U<var>                    Undefine preprocessor define
```

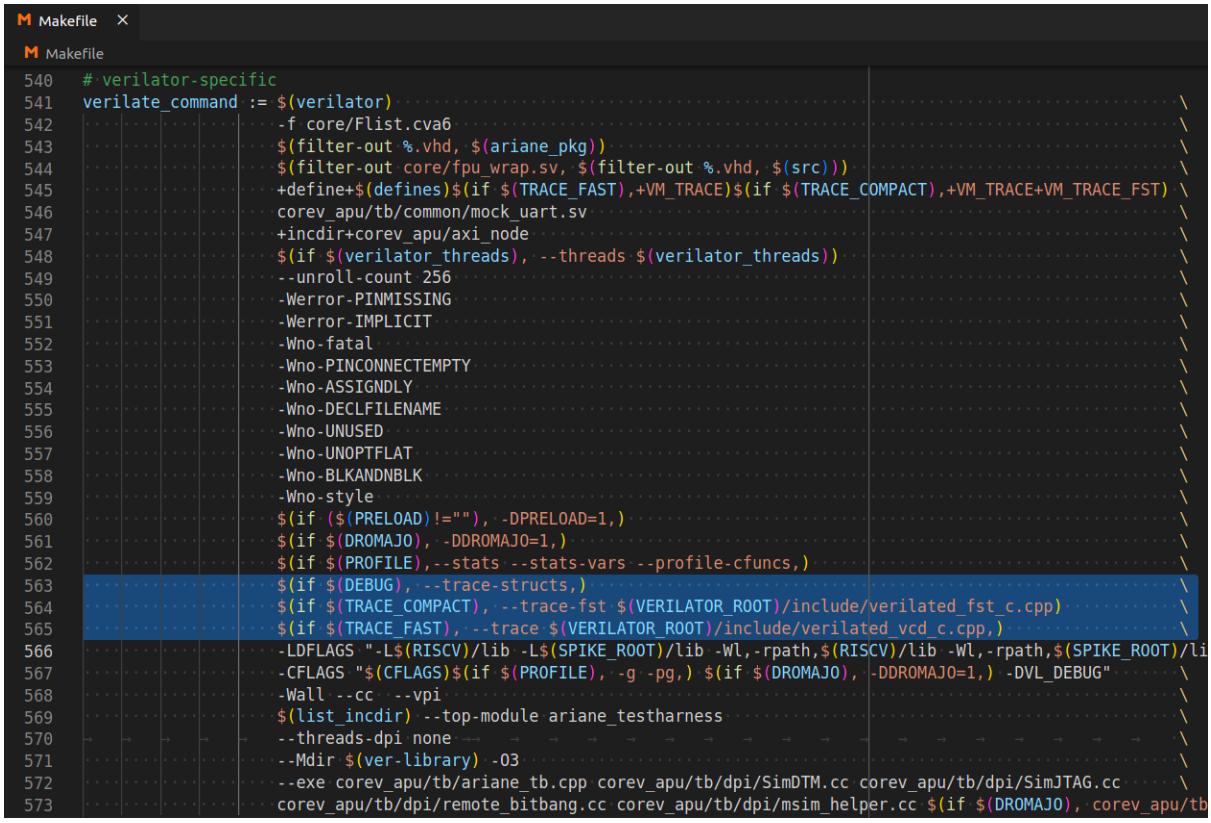
From [verilator's manpages]:

```
--top-module <topname>      Name of top level input module
--trace                     Enable waveform creation
--trace-depth <levels>      Depth of tracing
--trace-underscore          Enable tracing of _signals
 -U<var>                    Undefine preprocessor define
```

From our working version of verilator, downloaded via CVA6's installation scripts:

```
asier@asier-ThinkPad-L380:~$ verilator --version
Verilator 4.110 2021-02-25 rev UNKNOWN.REV
asier@asier-ThinkPad-L380:~$ verilator --help | grep -e --trace
Wide character in print at /usr/share/perl/5.30/Pod/Text.pm line 303.
        --trace                     Enable waveform creation
        --trace-coverage            Enable tracing of coverage
        --trace-depth <levels>      Depth of tracing
        --trace-fst                 Enable FST waveform creation
        --trace-max-array <depth>   Maximum bit width for tracing
        --trace-max-width <width>   Maximum array depth for tracing
        --trace-params              Enable tracing of parameters
        --trace-structs             Enable tracing structure names
        --trace-threads <threads>   Enable waveform creation on separate threads
        --trace-underscore          Enable tracing of _signals
```

From the above figures we can observe how, somewhere in the project's files, we should find some "`--trace*`" options that allow the processor to generate waveform files. It is not necessary to understand the meaning of all the options, they just give us a starting point to look around the project's code and understand what is going on under the hood. Indeed, the DEBUG Makefile option of the CVA6 processor repository points to such options. Remember, the "core-v-verif" repository pulls the RTL code from the "standard" CVA6 repository, so any possible changes or modifications to the "standard" repo should later be compatible.



The above figure shows an unseemly “verilate_command” variable using some parameters for verilator. This is the only place in CVA6’s Makefile (or any other file in the repo, for that matter) that such parameters are used, so it is safe to assume that these are the only three lines we should be caring about: the DEBUG line, the TRACE_COMPACT line and the TRACE_FAST line. A few modifications are needed to make these work properly, but let’s look at their meaning first.

Per the “`--help`” message of verilator:

```
--trace-structs
    Enable tracing to show the name of packed structure, union, and
    packed array fields, rather than a single combined packed bus. Due
    to VCD file format constraints this may result in significantly
    slower trace times and larger trace files.
```

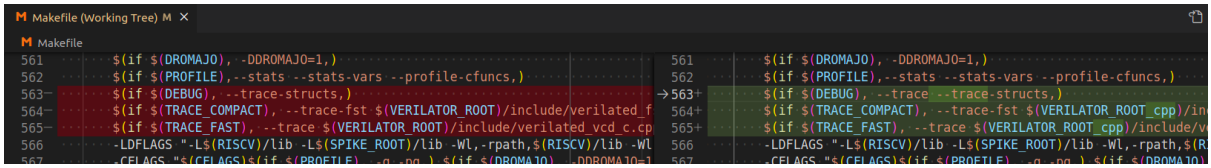
This means that, for the DEBUG option, any “std_logic_vector” in the RTL will be packed not only as a bus/grouping of bits, but also every bit of it with a corresponding signal value in the waveform file. Thus, every bit of every “std_logic_vector” is traced separately and any transition of any bit will be visible in a waveform viewer. The TRACE_FAST option is similar to the DEBUG option, but “std_logic_vectors” will be registered as standalone signals and not per-bit as with the “`--trace-structs`” flag enabled. The output waveform file of these two options comes in .vcd format. The verilated processor model admits both .vcd and .fst file formats:

```
asier@asier-ThinkPad-L380:~/cva6$ work-ver/Variane_testharness --help
Usage: work-ver/Variane_testharness [EMULATOR OPTION]... [VERILOG PLUSARG]... [HOST OPTION]... BINARY [TARGET OPTION]...
Run a BINARY on the Ariane emulator.

Mandatory arguments to long options are mandatory for short options too.

EMULATOR OPTIONS
  -r, --rbb-port=PORT      Use PORT for remote bit bang (with OpenOCD and GDB)
                           If not specified, a random port will be chosen
                           automatically.
  -v, --vcd=FILE,          Write vcd trace to FILE (or '-' for stdout)
  -f, --fst=FILE,          Write fst trace to FILE
                           Print performance statistic at end of test

HOST OPTIONS
  -h, --help               Display this help and exit
```

This .fst file format is what the TRACE_COMPACT line refers to, as .fst files are much smaller and more manageable than .vcd files. In fact, the DEBUG and TRACE_FAST options will generate 15/40GB-sized files. These are nearly-unmanageable file sizes, specially if one desires to generate a waveform file for each of the verification testbenches the “core-v-verif” repository executes, so the .fst file format is a wiser option in this case.

In order to reduce the file sizes of the .vcd files, one can opt to omit signals from specific modules or even omit signals at module-level, meaning that modules n-levels deep from the top module are not included in the waveform file. See the following sources (1, 2, 3, 4, 5) for more information on the matter, as well as lines 52-56 of file “tb_top.sv” in the “cva6/corev_apu/riscv-dbg/tb/” directory of the CVA6-specific repository to see how/where the $dumpvars() call is being made. We played around with the “`/*verilator tracing_off*/`” metacomment to omit some modules’ signals, but didn’t manage to obtain the desired result, as signals were being added to the .vcd/.fst files nonetheless. The “`--trace-depth`” option is also useful to a point. If an additional module must be added at a particularly low level module of the processor, nothing can be accomplished by playing around with this setting. Regardless, .fst files work wonderfully with GTKWave and take up very little space, so they ended up being our waveform-file-format of choice.

In order to make any of the options work, the following changes must be made to the Makefile of the project:



- The VERILATOR_ROOT environment variable must be changed to VERILATOR_ROOT_cpp (or some other name) to avoid conflicts with a “core-v-verif”-generated environment variable of the same name, pointing to the verilator installation in the “core-v-verif/tools/” directory. Declaring a VERILATOR_ROOT_cpp environment variable in .bashrc as “`export VERILATOR_ROOT_cpp=/usr/share/verilator`” should suffice.

```
asier@asier-ThinkPad-L380:~$ grep -r -C 5 VERILATOR_ROOT core-v-verif/cva6/README.md

The default values are:

- `RISCV_GCC`: `$RISCV/bin/riscv-none-elf-gcc`
- `RISCV_OBJCOPY`: `$RISCV/bin/riscv-none-elf-objcopy`
- `VERILATOR_ROOT`: `../tools/verilator-4.110` to install in core-v-verif/tools
- `SPIKE_ROOT`: `../tools/spike` to install in core-v-verif/tools

- `CVA6_REPO`: `https://github.com/openhwgroup/cva6.git`
- `CVA6_BRANCH`: `master`
- `CVA6_HASH`: see value in `regress/install-cva6.sh`
asier@asier-ThinkPad-L380:~$ echo $VERILATOR_ROOT_cpp
/usr/share/verilator
asier@asier-ThinkPad-L380:~$
```

- The "`--trace`" option must precede the "`--trace-structs`" flag in the DEBUG line for it to work properly
    - If not set, the verilated processor model will not be able to understand the "`--vcd`" flag option it supposedly supports (see figures below)

```
asier@asier-ThinkPad-L380:~/cva6$ grep "(DEBUG)" Makefile
                  $(if $(DEBUG), --trace-structs,)                                                  \
asier@asier-ThinkPad-L380:~/cva6$ make -j8 verilate DEBUG=1 &> somelogfile.log
asier@asier-ThinkPad-L380:~/cva6$ echo '
> #include <stdio.h>
>
> int main(int argc, char const *argv[]) {
>     printf("Hello CVA6!\n");
>     return 0;
> }' > hello.c
asier@asier-ThinkPad-L380:~/cva6$ riscv64-unknown-elf-gcc hello.c -o hello.elf
asier@asier-ThinkPad-L380:~/cva6$ work-ver/Variane_testharness --vcd=dump.vcd /riscvstuff/riscv64-unknown-elf/bin/pk hello.elf
work-ver/Variane_testharness: unrecognized option '--vcd=dump.vcd'
Usage: work-ver/Variane_testharness [EMULATOR OPTION]... [VERILOG PLUSARG]... [HOST OPTION]... BINARY [TARGET OPTION]...
Run a BINARY on the Ariane emulator.

Mandatory arguments to long options are mandatory for short options too.

EMULATOR OPTIONS
  -r, --rbb-port=PORT      Use PORT for remote bit bang (with OpenOCD and GDB)
                           If not specified, a random port will be chosen
                           automatically.

EMULATOR DEBUG OPTIONS (only supported in debug build -- try `make debug`)
  -v, --vcd=FILE,          Write vcd trace to FILE (or '-' for stdout)
  -f, --fst=FILE,          Write fst trace to FILE
  -p,                      Print performance statistic at end of test
```

Failed hello-world execution due to "`--vcd`" option not being recognized (see faulty DEBUG *line at the top)*

```
asier@asier-ThinkPad-L380:~/cva6$ grep "(DEBUG)" Makefile
                  $(if $(DEBUG), --trace --trace-structs,)                                          \
asier@asier-ThinkPad-L380:~/cva6$ make -j8 verilate DEBUG=1 &> somelogfile.log
asier@asier-ThinkPad-L380:~/cva6$ echo '
> #include <stdio.h>
>
> int main(int argc, char const *argv[]) {
>     printf("Hello CVA6!\n");
>     return 0;
> }' > hello.c
asier@asier-ThinkPad-L380:~/cva6$ riscv64-unknown-elf-gcc hello.c -o hello.elf
asier@asier-ThinkPad-L380:~/cva6$ work-ver/Variane_testharness --vcd=dump.vcd /riscvstuff/riscv64-unknown-elf/bin/pk hello.elf
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 35207
Starting VCD waveform dump ...
```

Successful hello-world execution (see correct DEBUG *line at the top)*

Introducing these two changes should result in trace files being generated for, in this case, the execution of a helloworld program in C compiled with the riscv-gnu-toolchain gcc compiler. Below are the timing and file-size results of executing the helloworld with the three possible options of DEBUG, TRACE_COMPACT and TRACE_FAST, as well as GTKWave opening a .fst file with no issue and doing the same process with a 12GB .vcd file:

```
asier@asier-ThinkPad-L380:~/cva6$ time make -j8 verilate DEBUG=1 &> somelogfile.log

real    2m53,161s
user    17m25,276s
sys     0m21,254s
asier@asier-ThinkPad-L380:~/cva6$ time work-ver/Variane_testharness --vcd=dump.vcd /riscvstuff/riscv64-unknown-elf/bin/pk hello.elf
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 36005
Starting VCD waveform dump ...
bbl loader
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:75: Verilog $finish
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:76: Verilog $finish
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:76: Second verilog $finish, exiting

real    10m56,394s
user    10m39,156s
sys     0m13,270s
asier@asier-ThinkPad-L380:~/cva6$ ll -h dump.vcd
-rw-rw-r-- 1 asier asier 12G dic 28 13:29 dump.vcd
asier@asier-ThinkPad-L380:~/cva6$
```

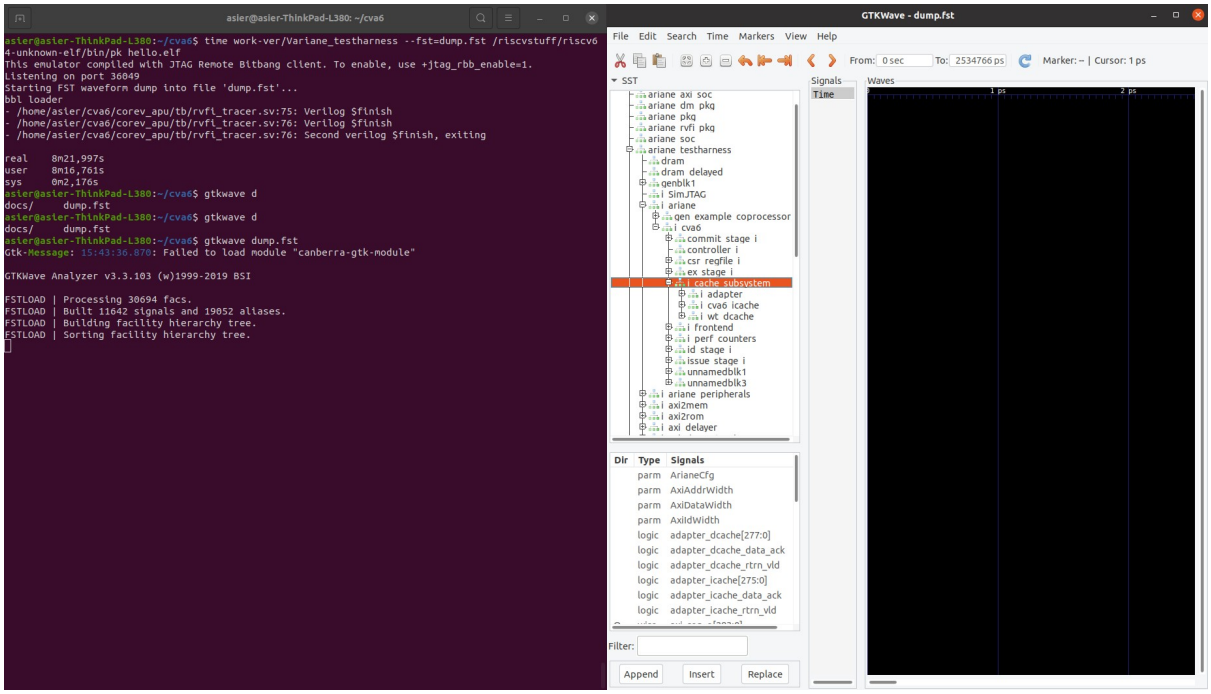
DEBUG *option in action*

```
asier@asier-ThinkPad-L380:~/cva6$ time make -j8 verilate TRACE_FAST=1 &> somelogfile.log

real    2m41,312s
user    14m32,659s
sys     0m17,521s
asier@asier-ThinkPad-L380:~/cva6$ time work-ver/Variane_testharness --vcd=dump.vcd /riscvstuff/riscv64-unknown-elf/bin/pk hello.elf
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 38673
Starting VCD waveform dump ...
bbl loader
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:75: Verilog $finish
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:76: Verilog $finish
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:76: Second verilog $finish, exiting

real    8m1,561s
user    7m19,539s
sys     0m32,834s
asier@asier-ThinkPad-L380:~/cva6$ ll -h d
docs/     dump.vcd
asier@asier-ThinkPad-L380:~/cva6$ ll -h dump.vcd
-rw-rw-r-- 1 asier asier 39G dic 28 13:13 dump.vcd
asier@asier-ThinkPad-L380:~/cva6$
```

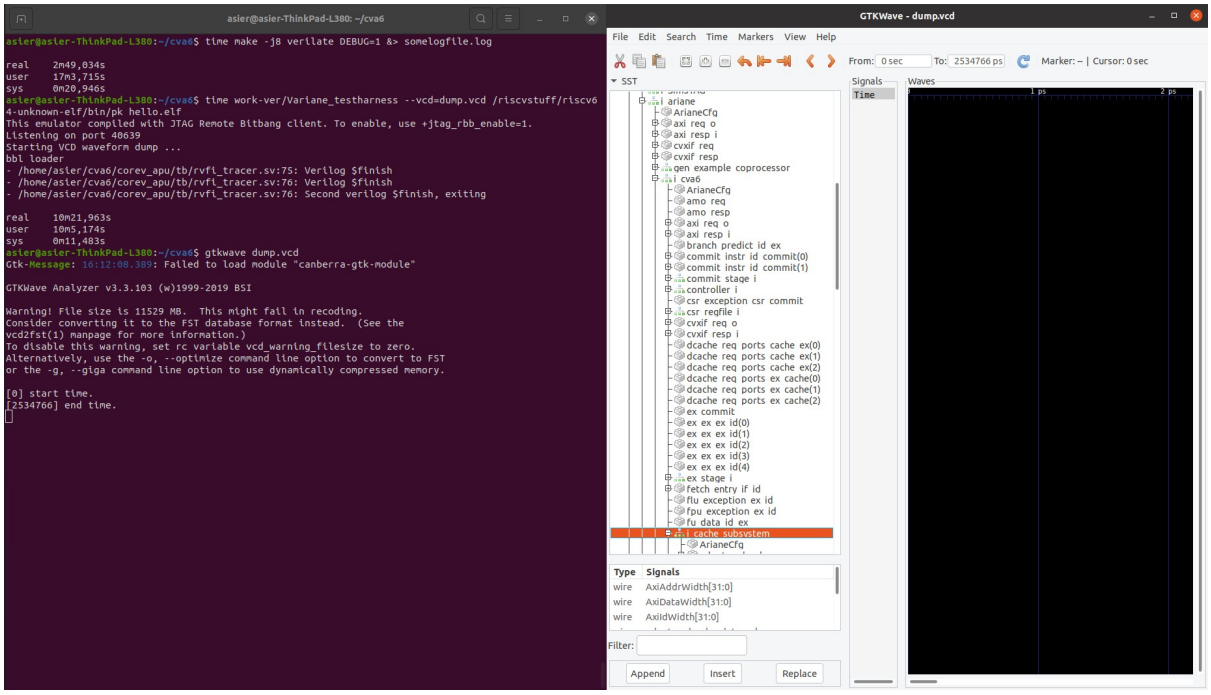
TRACE_FAST *option in action*

```
asier@asier-ThinkPad-L380:~/cva6$ time make -j8 verilate TRACE_COMPACT=1 &> somelogfile.log

real    2m29,572s
user    14m16,909s
sys     0m17,886s
asier@asier-ThinkPad-L380:~/cva6$ time work-ver/Variane_testharness --fst=dump.fst /riscvstuff/riscv64-unknown-elf/bin/pk hello.elf
This emulator compiled with JTAG Remote Bitbang client. To enable, use +jtag_rbb_enable=1.
Listening on port 41391
Starting FST waveform dump into file 'dump.fst'...
bbl loader
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:75: Verilog $finish
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:76: Verilog $finish
- /home/asier/cva6/corev_apu/tb/rvfi_tracer.sv:76: Second verilog $finish, exiting

real    8m15,953s
user    8m10,604s
sys     0m2,225s
asier@asier-ThinkPad-L380:~/cva6$ ll -h dump.vcd
ls: cannot access 'dump.vcd': No such file or directory
asier@asier-ThinkPad-L380:~/cva6$ ll -h dump.fst
-rw-rw-r-- 1 asier asier 252M dic 28 13:48 dump.fst
asier@asier-ThinkPad-L380:~/cva6$
```

TRACE_COMPACT *option in action*



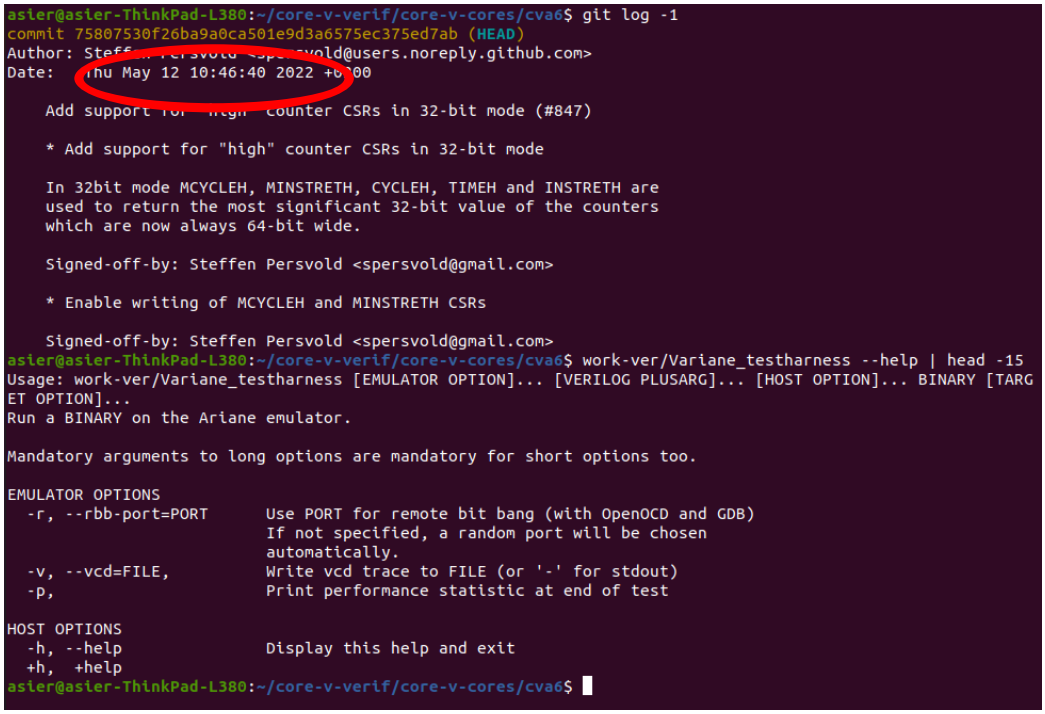
GTKWave opening the .fst file with no issues



GTKWave outputting some warnings about file sizes and recommending .fst file formats, even for 12GB-size files, and even logging the start-end times for loading the waveform file (there are, also, many more analyzable elements in the .vcd file picked up by GTKWave)

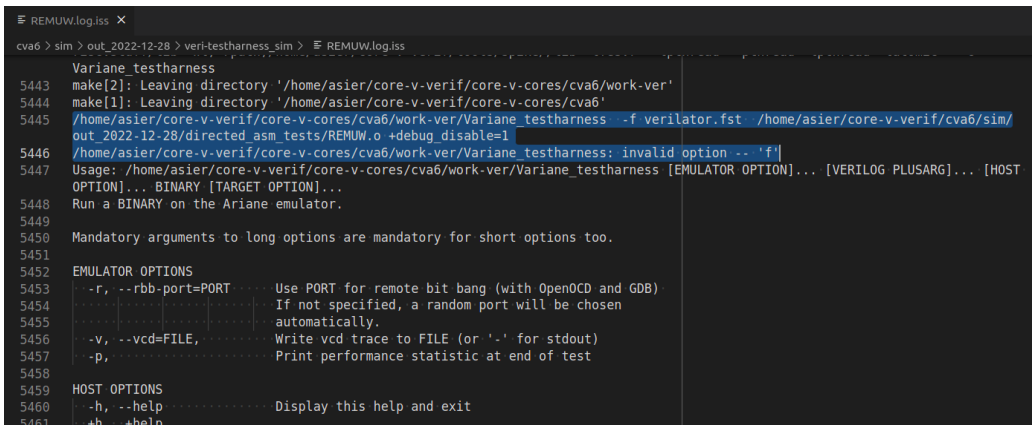
Waveforms in the “core-v-verif” repository

An interesting aspect of the “core-v-verif” repository is that the specific commit that it uses for the RTL of the CVA6 Ariane processor is way back from May of 2022, and, apparently, it does not support .fst waveform file generation (see images below).



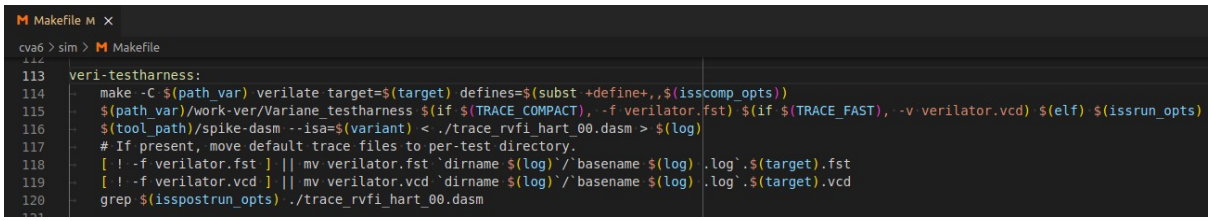
(1) Last commit from May of 2022 and (2) the verilated model of the processor does not mention the "`--fst`"option in the "`--help`" message (this verilated model can be generated by running either the "dv-riscv-compliance.sh" script or the "dv-riscv-tests.sh" script)

Trying to manually enter the `TRACE_COMPACT` flag into the "`make verilate`" command of the "core-v-verif" repository for running all the tests associated with the CVA6 processor results in the verilated model failing due to it not supporting the "`-f/--fst`" options, as expected from the contents of the previous figure. The following figure shows the log output into the REMUW.log.iss test log saying how, somewhere, the verilated processor model has been called with the "`-f verilator.fst`" option and that "`-f`" is not recognized as a valid flag for the model.



In order to obtain this output, we must dive back into the contents of the "core-v-verif" repository's Makefile for the CVA6 Ariane processor in the "core-v-verif/cva6/sim/" directory. In fact, some of the figures from the previous "core-v-verif" section of the document should

be making a lot more sense now. The [veri-testharness]-related figures from the previous section can be understood by looking at the first two lines of this Makefile command, so let's take a look at them:



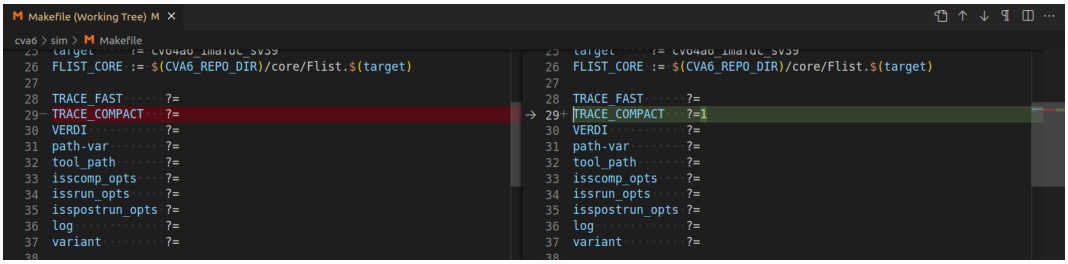
The first line corresponds, clearly, to the "`make verilate`" command necessary to compile the verilated model of the processor. This implies that the "core-v-verif" repository is following the same steps as if we were following them ourselves. The second line is, as expected, the actual call to the verilated model, the "Variane_testharness" file with the, surprise surprise, "`-f`" option for .fst waveform file generation baked into it. The last lines manage the .fst waveform files and neatly organise them according to the testbench being executed at the time.

There are a couple of caveats to claiming that the "core-v-verif" repository performs the same operations as if we were doing it manually in that the actual "core-v-verif" Makefile shown above is called through, first, the "dv-riscv-compliance.sh"/"dv-riscv-tests.sh" script (either of them), then through the "cva6.py" file, and then through some hardcoded Makefile commands in a "cva6.yaml" file, but all the essentials are here to make some modifications.

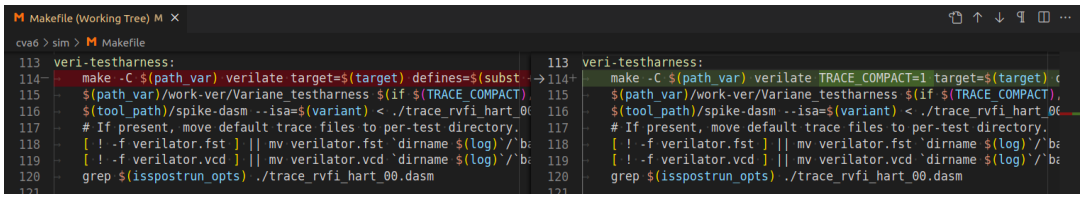
First of all, since the default commit for the CVA6 repository does not support .fst files (this being an automatic disqualification, unless one can afford to store many multiple-GB-sized files in their system), we'll remove this repository from the RTL directory altogether. In fact, following similar steps to what the "install-cva6.sh" script actually does without checkouting the specific "supported" commit of CVA6 should do the trick. The steps are as follows:

```
cd core-v-verif/core-v-cores/
rm -rf cva6
git clone https://github.com/openhwgroup/cva6.git
cd cva6
mkdir tmp
git submodule update --init --recursive
```

This should clone an updated version of the core's RTL repository and now, the only modifications left are to do with the "core-v-verif" repository's Makefile for the CVA6 processor. Essentially, we have managed to take advantage of everything we know about the standalone CVA6 repository, applying it to this verification repository to gain some (necessary) insight into the verification testbenches. The `TRACE_COMPACT` variable within the Makefile in the "core-v-verif/cva6/sim/" directory should be set to 1 (see first figure). Then, the "veri-testharness" section should have its "`make verilate`" command adapted to create a waveform-generating model of the processor (see second figure).



First modification, `TRACE_COMPACT` *set to 1*



Second modification, `TRACE_COMPACT=1` *in the "*`make verilate`*" command*

With this, running either script for CVA6's verification generates .fst files as expected, both for the compliance script and the test script of CVA6 (**<u>the results for the normal and the waveform-enabled verification scripts will be uploaded to the project's repository!</u>**):

```
asier@asier-ThinkPad-L380:~/core-v-verif/cva6/sim$ grep "PASSED," outwav_rvcompliance/iss_regr.log
171 PASSED, 0 FAILED
asier@asier-ThinkPad-L380:~/core-v-verif/cva6/sim$ du -cha outwav_rvcompliance/ | grep fst | head -30
600K    outwav_rvcompliance/veri-testharness_sim/I-JALR-01.cv64a6_imafdc_sv39.fst
376K    outwav_rvcompliance/veri-testharness_sim/amomax_w.cv64a6_imafdc_sv39.fst
464K    outwav_rvcompliance/veri-testharness_sim/SRAW.cv64a6_imafdc_sv39.fst
372K    outwav_rvcompliance/veri-testharness_sim/amoor_w.cv64a6_imafdc_sv39.fst
460K    outwav_rvcompliance/veri-testharness_sim/ADDW.cv64a6_imafdc_sv39.fst
460K    outwav_rvcompliance/veri-testharness_sim/I-SLTI-01.cv64a6_imafdc_sv39.fst
308K    outwav_rvcompliance/veri-testharness_sim/simple.cv64a6_imafdc_sv39.fst
600K    outwav_rvcompliance/veri-testharness_sim/bge.cv64a6_imafdc_sv39.fst
500K    outwav_rvcompliance/veri-testharness_sim/I-LH-01.cv64a6_imafdc_sv39.fst
468K    outwav_rvcompliance/veri-testharness_sim/I-SRLI-01.cv64a6_imafdc_sv39.fst
536K    outwav_rvcompliance/veri-testharness_sim/move.cv64a6_imafdc_sv39.fst
436K    outwav_rvcompliance/veri-testharness_sim/C-XOR.cv64a6_imafdc_sv39.fst
380K    outwav_rvcompliance/veri-testharness_sim/I-CSRRCI-01.cv64a6_imafdc_sv39.fst
464K    outwav_rvcompliance/veri-testharness_sim/MULW.cv64a6_imafdc_sv39.fst
636K    outwav_rvcompliance/veri-testharness_sim/I-BGE-01.cv64a6_imafdc_sv39.fst
628K    outwav_rvcompliance/veri-testharness_sim/I-BNE-01.cv64a6_imafdc_sv39.fst
584K    outwav_rvcompliance/veri-testharness_sim/srai.cv64a6_imafdc_sv39.fst
376K    outwav_rvcompliance/veri-testharness_sim/C-SW.cv64a6_imafdc_sv39.fst
576K    outwav_rvcompliance/veri-testharness_sim/lh.cv64a6_imafdc_sv39.fst
392K    outwav_rvcompliance/veri-testharness_sim/I-NOP-01.cv64a6_imafdc_sv39.fst
344K    outwav_rvcompliance/veri-testharness_sim/sbreak.cv64a6_imafdc_sv39.fst
432K    outwav_rvcompliance/veri-testharness_sim/ADDIW.cv64a6_imafdc_sv39.fst
640K    outwav_rvcompliance/veri-testharness_sim/rvc.cv64a6_imafdc_sv39.fst
344K    outwav_rvcompliance/veri-testharness_sim/I-CSRRWI-01.cv64a6_imafdc_sv39.fst
632K    outwav_rvcompliance/veri-testharness_sim/I-BLTU-01.cv64a6_imafdc_sv39.fst
444K    outwav_rvcompliance/veri-testharness_sim/I-CSRRC-01.cv64a6_imafdc_sv39.fst
468K    outwav_rvcompliance/veri-testharness_sim/I-XORI-01.cv64a6_imafdc_sv39.fst
496K    outwav_rvcompliance/veri-testharness_sim/I-SRA-01.cv64a6_imafdc_sv39.fst
468K    outwav_rvcompliance/veri-testharness_sim/I-SLLI-01.cv64a6_imafdc_sv39.fst
524K    outwav_rvcompliance/veri-testharness_sim/I-SH-01.cv64a6_imafdc_sv39.fst
asier@asier-ThinkPad-L380:~/core-v-verif/cva6/sim$
```

CVA6 compliance waveform files

```
asier@asier-ThinkPad-L380:~/core-v-verif/cva6/sim$ grep "PASSED," outwav_rvtests/iss_regr.log
117 PASSED, 0 FAILED
223 PASSED, 0 FAILED
asier@asier-ThinkPad-L380:~/core-v-verif/cva6/sim$ du -cha outwav_rvtests/ | grep fst | head -30
6,6M    outwav_rvtests/veri-testharness_sim/amomax_w.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amoor_w.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amomin_d.cv64a6_imafdc_sv39.fst
4,1M    outwav_rvtests/veri-testharness_sim/simple.cv64a6_imafdc_sv39.fst
4,3M    outwav_rvtests/veri-testharness_sim/bge.cv64a6_imafdc_sv39.fst
4,2M    outwav_rvtests/veri-testharness_sim/remw.cv64a6_imafdc_sv39.fst
4,8M    outwav_rvtests/veri-testharness_sim/move.cv64a6_imafdc_sv39.fst
4,3M    outwav_rvtests/veri-testharness_sim/srai.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/lh.cv64a6_imafdc_sv39.fst
348K    outwav_rvtests/veri-testharness_sim/sbreak.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amomax_d.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amominu_d.cv64a6_imafdc_sv39.fst
544K    outwav_rvtests/veri-testharness_sim/rvc.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amoswap_d.cv64a6_imafdc_sv39.fst
4,3M    outwav_rvtests/veri-testharness_sim/addiw.cv64a6_imafdc_sv39.fst
4,4M    outwav_rvtests/veri-testharness_sim/mulh.cv64a6_imafdc_sv39.fst
4,4M    outwav_rvtests/veri-testharness_sim/mul.cv64a6_imafdc_sv39.fst
4,4M    outwav_rvtests/veri-testharness_sim/add.cv64a6_imafdc_sv39.fst
368K    outwav_rvtests/veri-testharness_sim/ma_fetch.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/lhu.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amomaxu_w.cv64a6_imafdc_sv39.fst
6,6M    outwav_rvtests/veri-testharness_sim/amoadd_d.cv64a6_imafdc_sv39.fst
332K    outwav_rvtests/veri-testharness_sim/wfi.cv64a6_imafdc_sv39.fst
4,4M    outwav_rvtests/veri-testharness_sim/slt.cv64a6_imafdc_sv39.fst
4,4M    outwav_rvtests/veri-testharness_sim/subw.cv64a6_imafdc_sv39.fst
4,5M    outwav_rvtests/veri-testharness_sim/sllw.cv64a6_imafdc_sv39.fst
4,2M    outwav_rvtests/veri-testharness_sim/fclass.cv64a6_imafdc_sv39.fst
4,2M    outwav_rvtests/veri-testharness_sim/jalr.cv64a6_imafdc_sv39.fst
4,4M    outwav_rvtests/veri-testharness_sim/bltu.cv64a6_imafdc_sv39.fst
7,0M    outwav_rvtests/veri-testharness_sim/sw.cv64a6_imafdc_sv39.fst
asier@asier-ThinkPad-L380:~/core-v-verif/cva6/sim$
```

CVA6 tests waveform files (the RISC-V tests come in two sets, so that's why there are to total results of 117 and 223 sets, although the 223 is also counting the previous 117)

Installation script for the CVA6 repository

WARNING #1: some previously-specified changes to the repo's Makefile are necessary to create a waveform-generating verilated model of the processor, so proceed with due caution and at your own peril if your are jumping into this script without having read first everything we say in the above sections; **this installation script will also be added to the project's repository!**

WARNING #2: this is an adapted version of what the setup script of the standalone CVA6 repository should be doing; it works for us in our Ubuntu 20.04 systems and has been tried and tested a number of times to confirm that everything we show in this document is reproducible and doable, you shouldn't need to install anything else or follow any additional steps to get things working

```
# necessary .bashrc exports:
############################

# # CVA6-specific variables
# export RISCV=/riscvstuff
# export PATH=/riscvstuff/bin:$PATH
# export PATH=/riscvstuff:$PATH
# export NUM_JOBS=8
# export VERILATOR_ROOT_cpp=/usr/share/verilator

# # coreverif variables
# export CV_SW_TOOLCHAIN=/riscvstuff
# export CV_SW_PREFIX=riscv64-unknown-elf-
```

```
# # coreverif-cva6 variables
# export RISCV_GCC=$RISCV/bin/riscv64-unknown-elf-gcc
# export RISCV_OBJCOPY=$RISCV/bin/riscv64-unknown-elf-objcopy

# make absolutely sure envvars are loaded
cd
source .bashrc
# remove everything in root
sudo mkdir -p /riscvstuff
   # bear in mind that this is not the usual "/opt/riscv" directory, change
   # this and the RISCV envvar above if wanting to follow more customary
   # guidelines
sudo rm -rf /riscvstuff/*
sudo chmod 777 /riscvstuff

# remove stuff from home
# sudo rm -rf riscv*
   # this is necessary for removing externally-installed (to CVA6's repo)
   # toolchains and utils such as the gnu-toolchain or spike or the proxy
   # kernel
sudo rm -rf cva6/
sudo rm -rf core-v-verif/

# install some dependencies
sudo apt install -y git autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-
dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc
zlib1g-dev libexpat-dev
cd
echo 'deb
http://download.opensuse.org/repositories/home:/phiwag:/edatools/xUbuntu_20.04/ /' |
sudo tee /etc/apt/sources.list.d/home:phiwag:edatools.list
curl -fsSL
https://download.opensuse.org/repositories/home:phiwag:edatools/xUbuntu_20.04/
Release.key | gpg --dearmor | sudo tee /etc/apt/trusted.gpg.d/home_phiwag_edatools.gpg >
/dev/null
sudo apt update
sudo apt install verilator-4.110 device-tree-compiler

# onto Ariane's stuff
git clone https://github.com/openhwgroup/cva6.git
cd cva6
mkdir tmp
git submodule update --init --recursive
./ci/build-riscv-gcc.sh
   # comment out the "git checkout $VERSION" line, some submodules don't exist anymore
   # would advise to avoid having console output redirected to /dev/null
   # this is going to take a looooooong time
./ci/install-spike.sh
./ci/install-riscvpk.sh
./ci/build-riscv-tests.sh
   # would advise to avoid having console output redirected to /dev/null

# tutorial steps to do the helloworld
echo '
#include <stdio.h>

int main(int argc, char const *argv[]) {
   printf("Hello CVA6!\n");
   return 0;
}' > hello.c
riscv64-unknown-elf-gcc hello.c -o hello.elf

# # 10-15GB .vcd files
# make -j8 verilate DEBUG=1
# work-ver/Variane_testharness --vcd=dump.vcd /riscvstuff/riscv64-unknown-elf/bin/pk
hello.elf

# # 40-50GB .vcd files
# make -j8 verilate TRACE_FAST=1
# work-ver/Variane_testharness --vcd=dump.vcd /riscvstuff/riscv64-unknown-elf/bin/pk
hello.elf
```

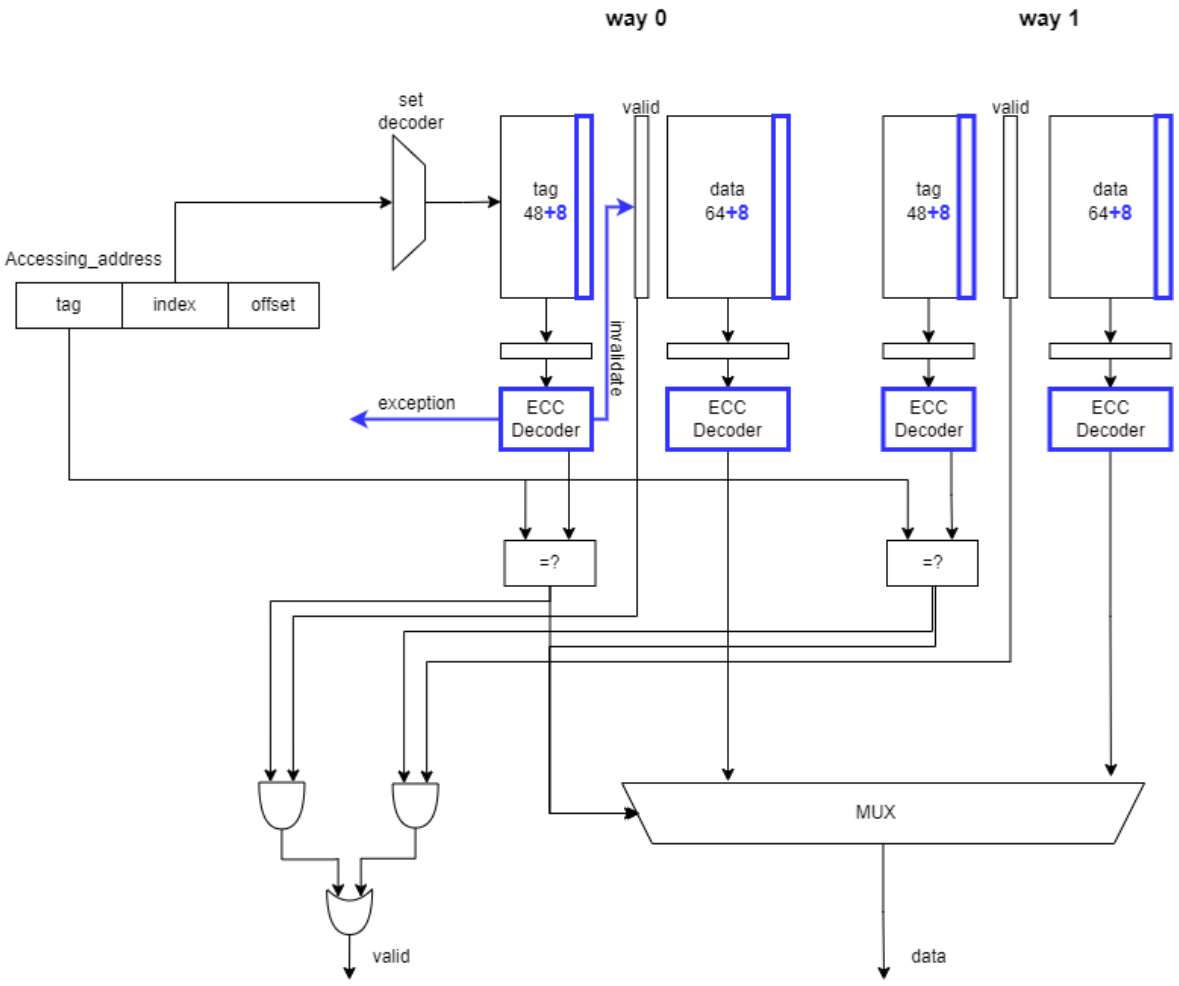
```
# 300MB .fst files
make -j8 verilate TRACE_COMPACT=1
work-ver/Variane_testharness --fst=dump.fst /riscvstuff/riscv64-unknown-elf/bin/pk
hello.elf
```

3. Theoretical integration and hurdles

Integration within a simplified cache structure

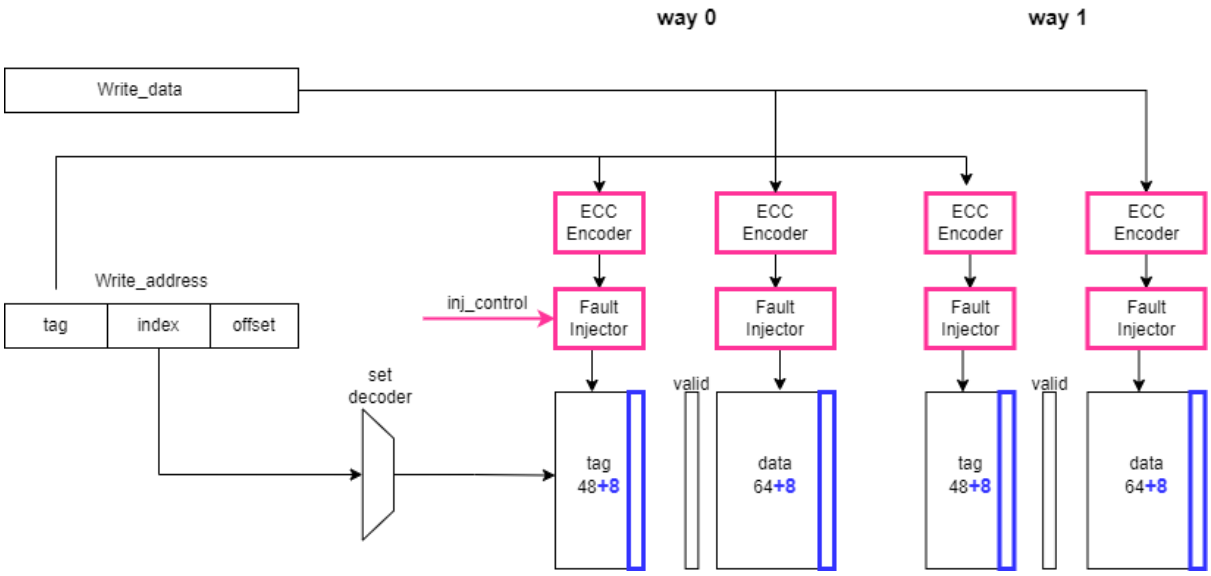
The default cache of the Ariane kernel is a maximum 8-way 32 KB associative cache with 16 byte line sizes. The default configuration of the Ariane data cache is write-back with no write allocation, although other configurations are available. To make it simpler and to show graphically how ECC is implemented in a cache, we will use a conventional 2-way cache.

The following diagram shows a conventional 2-way cache with ECC. In this case the parts of the cache added to enable the ECC function are shown in blue. The tag and data arrays have been extended to store the ECC correction bits. Also the ECC Decoder has been added to the output of the tag and data arrays (after the DFFs). We need as many decoders as the number of ways in the cache and these decoders will perform the functions described in previous labs to detect and correct errors. As we have seen in previous labs, in the case of finding an error there are several options: correct it without the system noticing it, and, in the case of double bit errors, either invalidate the array or throw an exception in the case of dirty data. These last two options need new signals, which have been drawn in blue in the diagram.



If any data structure within the cache (the tag arrays or data arrays or other) doesn't match with any of the supported data lengths of the [Hsiao70] ECC implementation, these fields would have to be extended to match the expected lengths of 16, 32 or 64 data bits, as is the case with the tag arrays in the example above (48 data bits + 8 ECC btis + the implicit 16 additional data bits set to '0' needed).

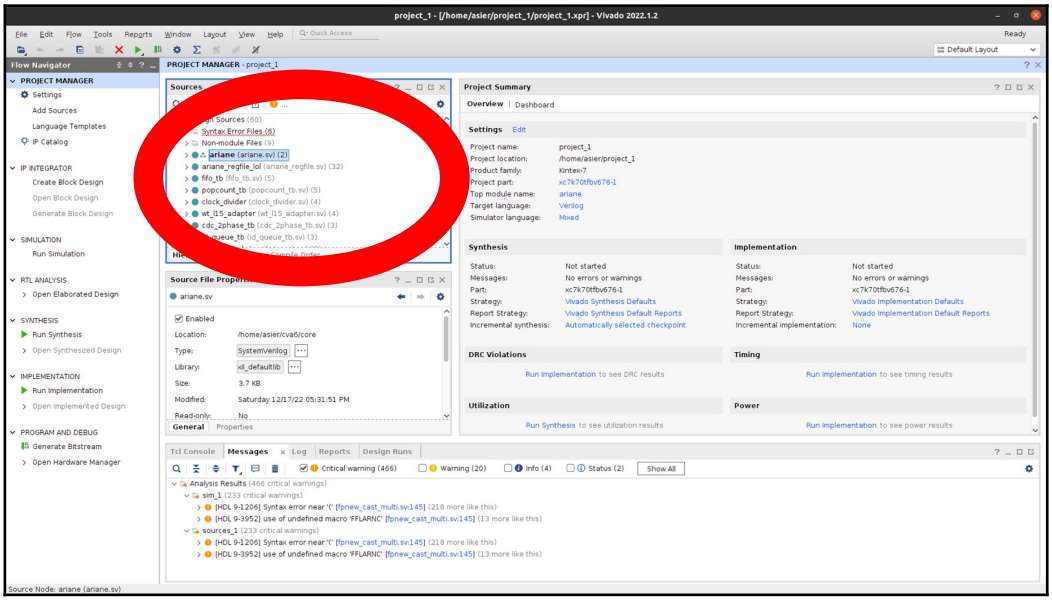
In other labs of this project, ECC Encoder and Fault Injector modules have been implemented to assist in the implementation and correction of ECC on Ariane. In a processor with ECC capabilites, the data coming from main memory comes already with ECC correction bits, but in our case, the Ariane processor does not implement ECC, so we generate the ECC correction bits at the input of the tag and data arrays. In the following diagram you can see in pink the modifications to be made to the processor to check the correct implementation of ECC. We can also see the Fault Injector module, already seen in other labs, used to force ECC errors and help the implementation evaluation. It incorporates a control signal that allows us to select if we want to generate no errors, 1 bit errors, 2 bit errors or randomly switching between the three previous options for data to have no errors, others have 1 bit errors and others have 2 bit errors.The Fault Injector module(s) are placed between the encoders and wherever data is residing, they can also be seen in pink in the diagram.



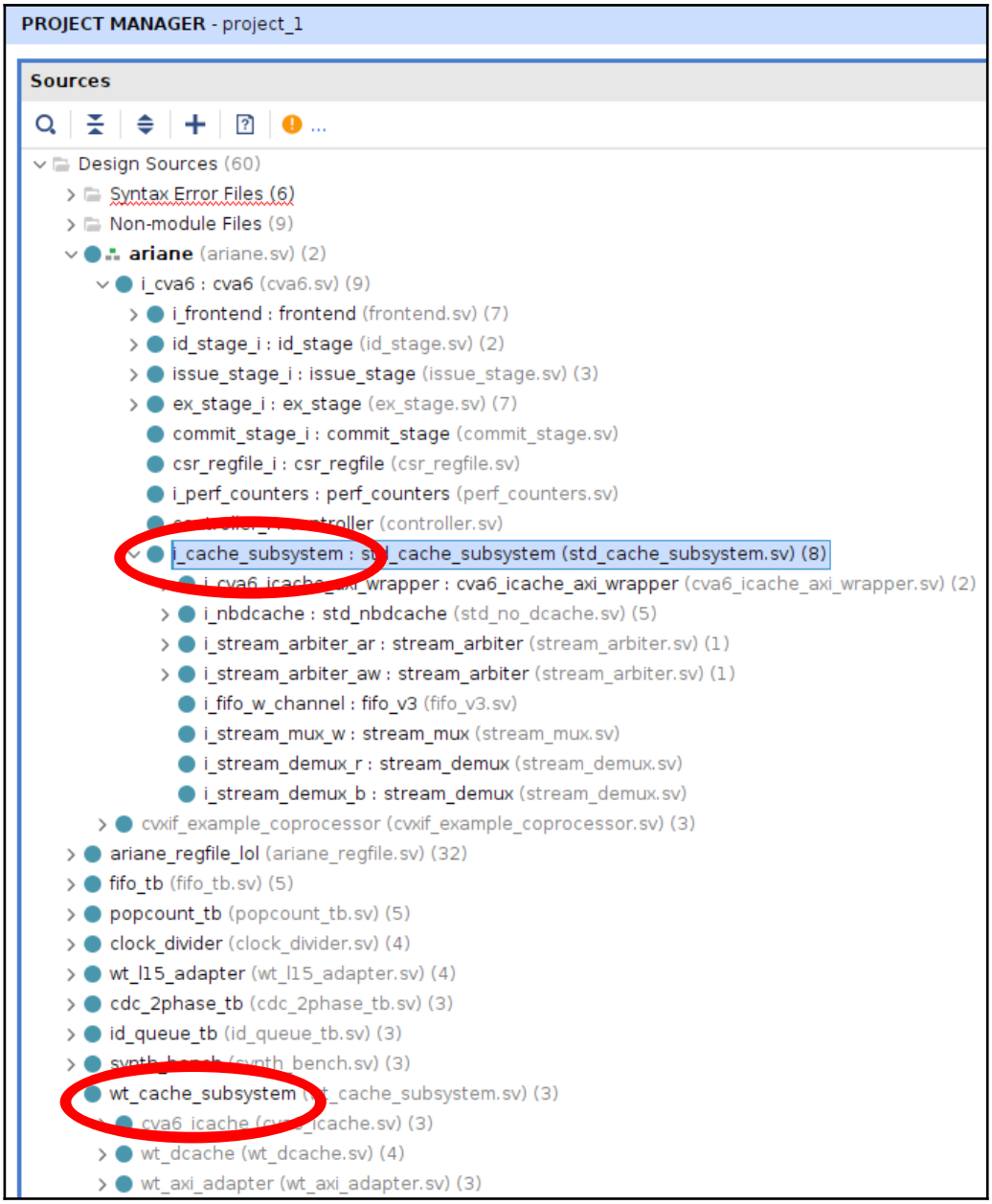
As we have seen, the greater the number of ways, the greater the number of logic modules to be used, both for Decoders and Encoders and fault injectors. Moreover, the instruction cache uses SED-DED and the data cache uses SEC-DED since it is (typically) not writeable. The configuration shown is the simplest of the many options that exist, there are several papers that show implementations that seek to improve performance (reduce cache latency), improve reliability (greater resistance to errors) or improve flexibility (combination of both in a configurable way). See this, for example.

Reality of the code

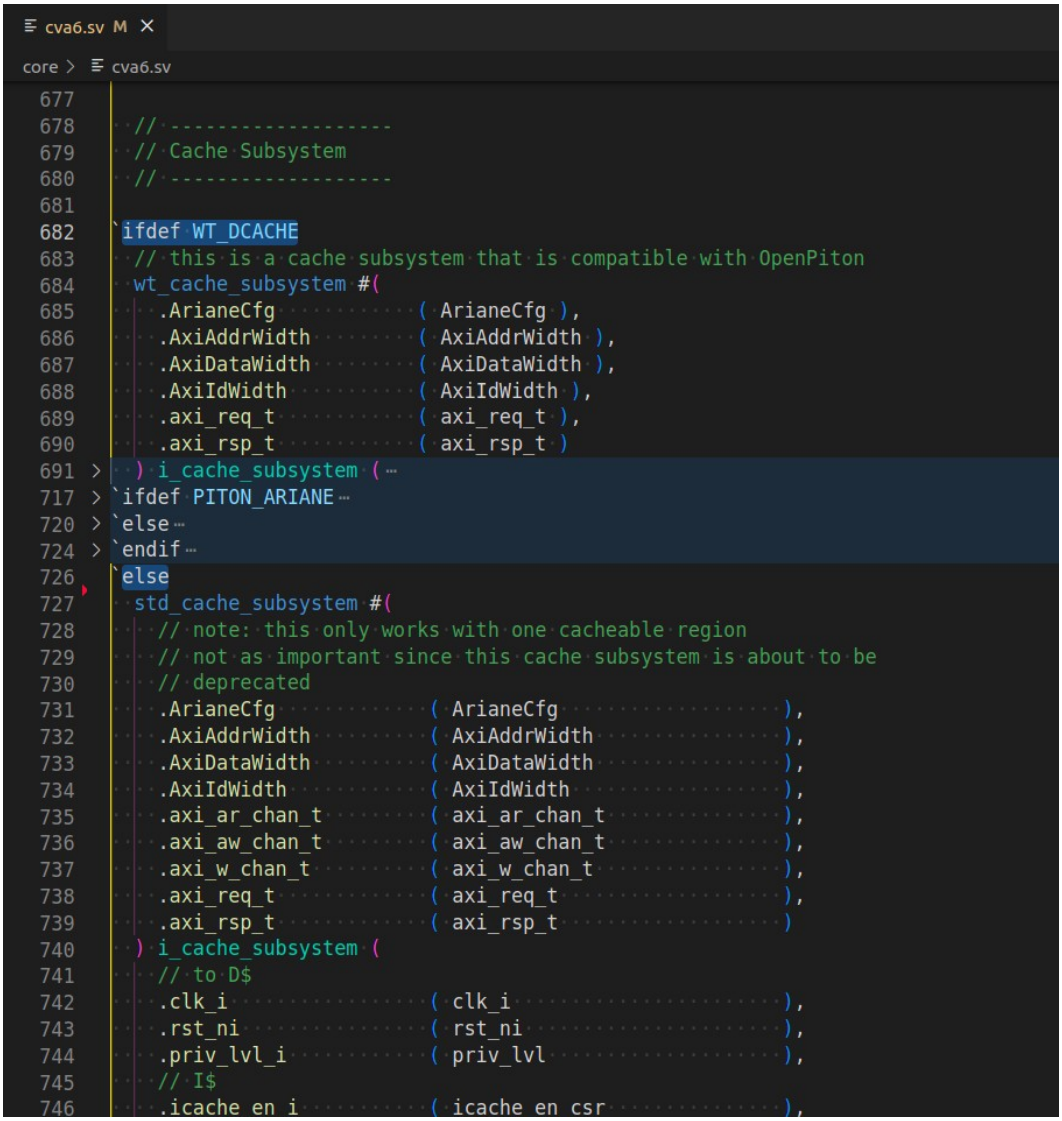
Vivado and its source code hierarchy tree has proven to be a particularly useful tool for determining the structure of the code and the RTL modules of the processor. It even goes as far as identifying existing modules that end up not being called/created by the top module due to compilation variables not having specific values.



Vivado project after importing all the RTL sources from the "cva/core/" directory and detecting a top-and-submodules structure to the RTL, as expected

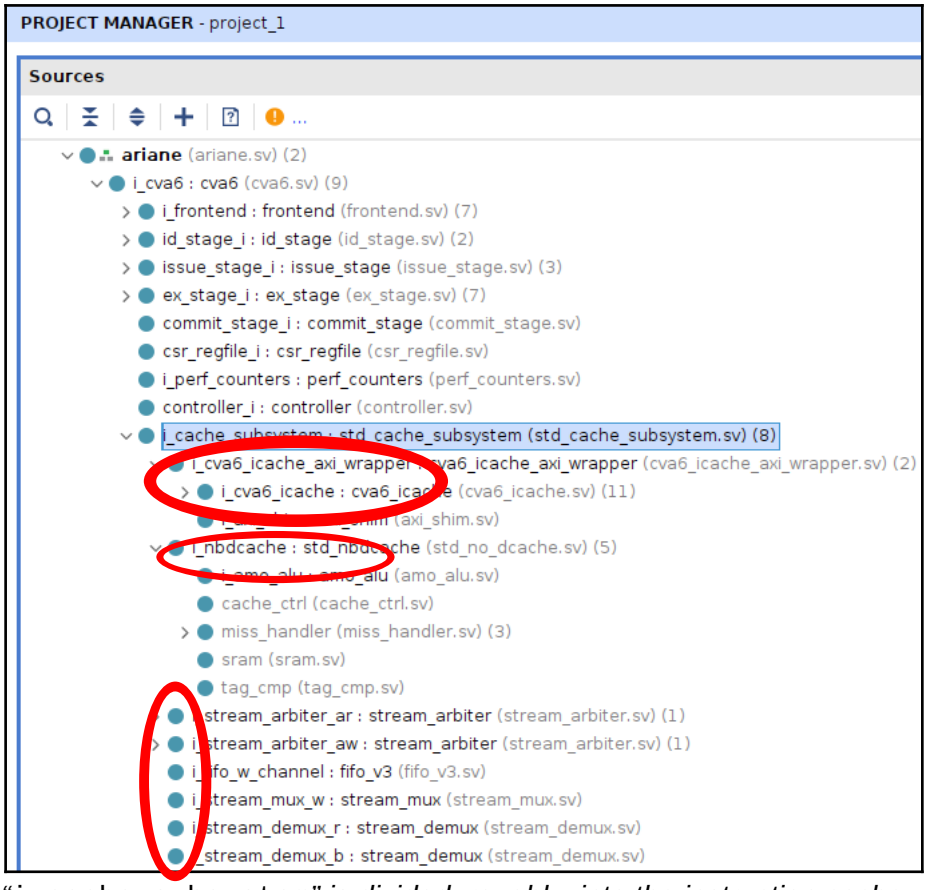


"`ariane`" top module and its submodules; notice the distinction between "`i_cache_subsystem`" and "`wt_cache_subsystem`" (see next code figure)



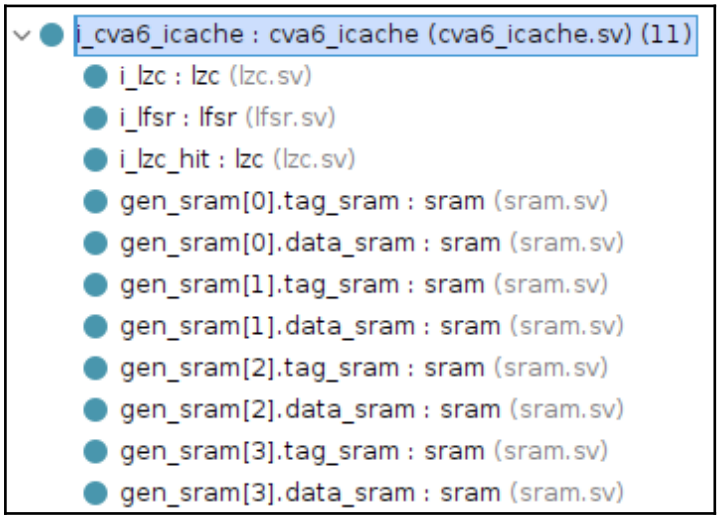
The "`i_cache_subsystem`*" module was correctly detected by the Vivado hierarchy tree as a "*`std_cache_subsystem`*" module and not the write-through-type*

Taking a closer look into the cache subsystem components, we can distinguish an instruction cache wrapped in some AXI-related components, a data cache in the "`i_nbdcache`" module managed by an atomic memory operations unit and some other intermediate modules. We were able to determine that "`i_nbdcache`" was referring to the data cache not only owing to its name, but also because of the signals mapped it in the "`std_cache_subsystem.sv`" file ("`dcache_enable_i`", "`dcache_flush_i`" and the like).



"`i_cache_subsystem`" is divided, roughly, into the instruction cache with its AXI wrapper, the data cache and some other intermediate modules

Delving a little deeper into the "`i_cva6_icache`" module within the "`cva6_icache.sv`" file reveals some more interesting information about how the instruction cache is organised into four distinct tag-data blocks.



However, that's about where the fun stops. The actual code of the "`cva6_icache.sv`", although it does come with some assertions at the end of the file for verifying the module in isolation, is convoluted enough as it is for us to have been able to parse through it and

understand it enough to determine where exactly our ECC modules should “go”. And that’s only taking into account the “`cva6_icache.sv`” file with its “`cva6_icache`” module, not even considering the other modules “on top” as potential candidates for modification, if at all needed (a fact which we ignore). Managing to obtain waveforms files from the “core-v-verif” repository was burdensome and time-consuming enough as it is to have had some more time for analysing the code further. We could include screenshots showing our general, surface-level understanding of how we intuit the code for the cache subsystem works, but the reality is that, at time of writing this document, any contribution we hoped to make in this regard would fall not too far from what a couple afternoons of uninterrupted code spelunking would garner. This is a major pain point for the output of our work for the project, but we hope that our investigations into the various repositories involved serves as a stepping stone of sorts that can save future students some time to actually tinker with the code that we haven’t had the pleasure of modifying into a respectable, working endeavour related to the PD subject.