gem5 Tests

To check that your changes to gem5 work, you should run some of our gem5 tests.

gem5 Test Categories

We run tests on the gem5 codebases regularly to ensure that changes do not break the code. These tests exist in 4 main categories;

1. **CPP Unit tests**: These are tests that run C++ code. In gem5, we use the Google Test framework.
2. **Python Unit tests**: These are tests that run Python code. In gem5, we use the Python unittest framework.
3. **TestLib Tests**: These are tests that run gem5 simulations, verify exit codes, and compare output to expected output ("testlib" is the name of the framework used to do this).
4. **Compilation Tests**: Tests which compile gem5 under different configurations with different compilers/environments.

Note: There are some tests we run which don't fit into these categories, but these are the main ones.

gem5 Test Schedule

1. **CI Tests**: These tests are run on every pull request to gem5, and every update to any pull request. The CI tests consist of the CPP and Python unit tests and a subset of the TestLib tests and Compilation tests.
   These are designed to run "quickly" (by gem5 standards), in under 4 hours.
2. **Daily Tests**: These tests are run every day on the gem5 codebase.
   These tests consist of the larger Testlib tests.
   They typically take 12 hours or more to complete.
3. **Weekly Tests**: These tests are run weekly on the gem5 codebase.
   These tests consist of the largest Testlib test suite and the Compilation tests.
   These tests typically take 1 to 2 days to complete.
4. **Compiler Tests**: These tests are run every week.
   These run a cross product of gem5 compilation targets and compilers the project currently supports.
   These tests usually take around 12 hours to complete.

The complete GitHub Actions workflow for these tests can be found in the .github/workflows/ in the gem5 repository.

We not go over these in this session but you can look over these yaml files and see how GitHub Actions is triggered to run these the gem5 tests.

CPP Unit tests

src/base/bitfield.test.cc is a typical example of a CPP unit test in gem5.

It is a GTest. More information on GTest can be found at https://google.github.io/googletest/

The SConscript file in the same directory as the test file is used to build the test.

```
GTest('bitfield.test', 'bitfield.test.cc', 'bitfield.cc')
```

The format is `GTest(<test_name>, <test_source>, <source_files>)`.

Running the CPP Unit Tests

You can run all unit tests with `scons build/ALL/unittests.opt` command.

To run a specific test:

```
scons build/ALL/base/bitfield.test.opt
./build/ALL/base/bitfield.test.opt
```

Python Unit Tests

tests/pyunit/util/pyunit_convert_check.py is a typical example of a Python unit test in gem5.

More information on Python's unittest framework can be found at https://docs.python.org/3/library/unittest.html

The tests are run with `gem5 tests/run_pyunit.py` command.
In our case, any file in the "tests/pyunit" directory with the prefix "pyunit_" is considered a test by the test runner.

Individual subdirectories in 'tests/pyunit" can be specified and run separately by passing those subdirectories as arguments to "tests/run_pytests.py". E.g.: `gem5 tests/run_pyunit.py --directory tests/pyunit/util`.

Compiler Tests

Compiler tests are run weekly on the gem5 codebase.

These tests are specified directly in a GitHub Action workflow: .github/workflows/compilers-tests.yaml

These tests use a series of Docker images to test compilation of various gem5 conifgurations with different compilers.

TestLib Tests

TestLib tests are the most important tests in gem5.
These tests run gem5 simulations and verify the output of the simulation.
The tests are written in Python and use the "testlib" framework to run the simulations and verify the output.

The tests are run using the `./main.py` command in the `test` directory of the gem5 repository.

It's useful to just focus on a subdirectory of tests when running tests:

```
./main.py run gem5/memory
```

The above will only run the "quick" tests in the "tests/gem5/memory" directory.
The "quick" tests are the testlib tests run in the CI pipeline. To run the tests in the "daily" or "weekly" test suites, you can use the `--length` to specify `long` or `very-long` (`quick` is the `length` default).

The `./main.py list` command can be used to list all the tests in a directory, which we'll demonstrate here:

```
# List all the long tests in tests/gem5/memory: Those run in the Daily Tests.
./main.py list --length long gem5/memory

# lists all the very long tests in tests/gem5/memory: Those run in the Weekly Tests.
./main.py list --length very-long gem5/memory
```

How TestLib Tests are declared

Let's look at "tests/gem5/m5_util" to see how a test is declared.

In this directory there is "test_exit.py".
Any file with the prefix "test_" is considered a test by the testlib framework and will be run when the tests are executed.

"configs" is a directory of configuration scripts that are used to run the tests defined in "test_exit.py".

Now, let's look into "test_exit.py" and see how the tests are declared.

```
import re

# Import the testlib framework. This is required.
from testlib import *
```

```
# Here we define a regular expression to match the output of the simulation.

m5_exit_regex = re.compile(
    r"Exiting @ tick \d* because m5_exit instruction encountered"
)

# ...

# Here we define the verifier using the regular expression.
a = verifier.MatchRegex(m5_exit_regex)
```

```
gem5_verify_config(
    name="m5_exit_test", # The test name
    verifiers=[a], # The verifier (must be iterable!)
    fixtures=(),
    config=joinpath( # The path to the config file to run.
        config.base_dir,
        "tests",
        "gem5",
        "m5_util",
        "configs",
        "simple_binary_run.py",
    ),
    # ... continued on next slide...
```

```
    config_args=[ # The arguments to pass to the config file.
        "x86-m5-exit",
        "--resource-directory",
        resource_path,
    ],
    # The ISAs to run on this test. In this cases "ALL/gem5.opt" is used.
    # `constants.arg_tag`: "ARM/gem5.opt"
    # `constants.x86_tag`: "X86/gem5.opt"
    # `constants.riscv_tag`: "RISCV/gem5.opt"
    valid_isas=(constants.all_compiled_tag,),
)
```

While not specified directly we could determine whether the tests runs as `quick`, `long`, or `very-long` with a `length` which accepts `constants.quick_tag`, `constants.long_tag`, or `constants.very_long_tag` as arguments (default is `constants.quick_tag`).

View these tests with:

```
./main.py list gem5/m5_util
```

```
Loading Tests
Discovered 12 tests and 6 suites in /workspaces/gem5-bootcamp-2024/gem5/tests/gem5/m5_util/test_exit.py
==================================================
Listing all Test Suites.
==================================================
SuiteUID:tests/gem5/m5_util/test_exit.py:m5_exit_test-ALL-x86_64-opt
==================================================
Listing all Test Cases.
==================================================
TestUID:tests/gem5/m5_util/test_exit.py:m5_exit_test-ALL-x86_64-opt:m5_exit_test-ALL-x86_64-opt
TestUID:tests/gem5/m5_util/test_exit.py:m5_exit_test-ALL-x86_64-opt:m5_exit_test-ALL-x86_64-opt-MatchRegex
==================================================
```

Then run with

```
./main.py run gem5/m5_util
```

Note: This will try and build "ALL/gem5.opt" each time you run the tests.
This can be time consuming.
You can pre-build the ALL/gem5.opt build with `scons build/ALL/gem5.opt -j$(nproc)` then, when running `./main.py run gem5/m5_util` add the `--skip-build` flag to skip the build step: `./main.py run --skip-build gem5/m5_util`.

If you want/need to build at this step, pass `-j$(nproc)` to the `./main.py run` command.

Exercise: Creating a TestLib Test

Go to materials/06-Contributing/02-testing/01-testlib-example.

Move "01-testlib-example" to "tests/gem5/" in the gem5 repository.

Provided in "test_example.py" is the `gem5_verify_config` function which is used to define testlib tests.

```
gem5_verify_config(
    name="test-example-1", # Name of the test. Must be unique.
    verifiers=(), # Outside exit-code zero check, additional  to be added.
    fixtures=(), # Fixtures: this is largely deprecated and can be ignored.
    config=joinpath(), # The path to the config script.
    config_args=[], # The arguments to be passed to the config script.
    valid_isas=(constants.arm_tag), # Need to run on ARM ISA
    length=constants.quick_tag, # A quick test to run in the CI pipeline
)
```

In this exercise we will do the following:

1. Create a test that runs the `example_config.py` script without any arguments and verifies it runs correctly.
2. Have this test use the `--to-print` argument to print "Arm Simulation Completed." at the end of the simulation.
3. Update this test with a verifier that checks the output of the simulation after the run is complete.
4. Write a second test that does the same as the first test but with a different output message (inclusive of a verifier).

After each step run the tests to verify the changes:

```
./main.py run gem5/01-testing-example
```

Hints and tips

- Adding `-vvv` to the end of the test command will give you more information about the test, particularly if an error occurs.
- Look at the other tests in "tests/gem5" for examples of how to write tests.
- You can pre-build the ARM/gem5.opt build using

```
scons build/ARM/gem5.opt -j$(nproc)
```

Then, when running `./main.py run gem5/02-testing`, add the `--skip-build` flag to skip the build step.

Congratulations!

You now know how to create tests for gem5 and run them to verify your changes.

We strongly encourage you to write tests for any changes you make to gem5 to ensure that your changes work as expected and do not break any existing functionality.