YosysHQ AppNote-400

YosysHQ GmbH

Oct 02, 2023

CONTENTS

This guide will explain how to set up a project from scratch using the bit-counter project example `bitcnt`. You can find this example in the directory `examples/mcy_demo/bitcnt` of the Tabby CAD Suite or OSS CAD Suite download, or in the MCY source.

You should start out with the following files:

`bitcnt_tb.v`: a self-checking testbench (or a set of testbenches) for which you wish to measure coverage. This is not restricted to HDL testbenches but can be any kind of test that can be launched from a script without manual intervention and can return a PASS/FAIL result.

`bitcnt.v`: this represents the synthesizable design (or portion of the design) that is going to be mutated by MCY. This design can comprise multiple modules/files. If your design is large, or requires many steps of formal verification, you should split your design into multiple parts to be mutated separately. The "mutation top level module" does not have to be the same as the module under test in the testbench, it can be any module in its submodule hierarchy.

After completing this tutorial, your files should match the contents of the `bitcnt` project from the MCY examples directory.

CHAPTER

ONE

CONFIGURATION FILE

Create a new directory for your project, and inside it create a new file called `config.mcy`. This is the main MCY configuration file. There are several required sections in a MCY config file which will be progressively filled in in the next steps. Insert the first five headers for these sections now:

```
[options]

[script]

[files]

[logic]

[report]
```

The first setting can already be added now, under the `[options]` section:

```
[options]
size 10
```

This is the number of mutations that MCY will generate. To begin with, this very low is chosen number during setup of the project so that the test runs are quick. Once everything is running correctly, we will increase the size of the mutation set.

CHAPTER

TWO

MUTABLE NETLIST GENERATION

Yosys introduces mutations in a (coarse-grain) synthesized netlist. In a first step, we need to tell Yosys how to build this netlist.

In the `[script]` section, fill in the steps to read your design into Yosys:

```
[script]
read -sv bitcnt.v
prep -top bitcnt
```

This should be a series of `read` commands, one per file. If your design is large, include only the subset of files necessary to build the module that should be mutated. When all sources are read in, the line `prep -top bitcnt` runs coarse grain synthesis on the module to be mutated designated by `-top`.

You can test your script either in interactive mode in Yosys or by saving it in a file `script.ys` (without the `[script]` header) and running `yosys script.ys`.

In the `[files]` section, list the paths to the files required by the script:

```
[files]
bitcnt.v
```

CHAPTER

THREE

TESTBENCH SCRIPT

Create a file named `test_sim.sh`. This will run the existing testbench on the mutated design. Add a bash preamble, then call the script `create_mutated.sh`:

```
#!/bin/bash

exec 2>&1
set -ex

bash $SCRIPTS/create_mutated.sh
```

When MCY runs, the tests will be executed in a temporary directory `tasks/<uuid>`, so the paths should be relative to this location.

The script `create_mutated.sh` reads the mutation description files prepared in the temporary task directory by MCY and (if called with no additional arguments) writes a file `mutated.v` containing the mutated module.

Next, insert whatever code will run your testbench as usual, but replacing the original source for the module to be mutated (`bitcount.v`) with the mutated source (`mutated.v`). The bitcnt testbench uses `iverilog`:

```
iverilog -o sim ../../bitcnt_tb.v mutated.v
vvp -n sim > sim.out
```

Finally, write the status returned by the testbench to the file `output.txt`:

```
if grep PASS sim.out && ! grep ERROR sim.out; then
        echo "1 PASS" > output.txt
elif ! grep PASS sim.out && grep ERROR sim.out; then
        echo "1 FAIL" > output.txt
else
        echo "1 ERROR" > output.txt
fi

exit 0
```

The 1 before the status is the test index. For tests with significant setup costs, it is possible to test multiple mutations in a single execution, in which case this number identifies the test run. Here we run each test individually so the index is always 1.

You can test that this portion works correctly as follows:

- create the directories `database` and `tasks/test` inside the project directory

  Note: these directories will get deleted when you run MCY so do not save any important files in them.

- add `write_ilang database/design.il` to the end of the `script.ys` file created earlier

- run the following commands:

```
yosys script.ys
cd tasks/test
echo "1 mutate -mode none" > input.txt
SCRIPTS=/usr/local/share/mcy/scripts bash ../../test_sim.sh
```

  (Adjust the path for SCRIPTS to match the MCY install location if necessary.)

- verify that the file `output.txt` was created and contains `1 PASS`.

If everything is working, add the following section to the bottom of `config.mcy`:

```
[test test_sim]
expect PASS FAIL
run bash $PRJDIR/test_sim.sh
```

This tells MCY that the test `test_sim` exists and how to run it. If `output.txt` ever contains a value not listed under `expect` when this test is run, the entire MCY process will be aborted.

CHAPTER

FOUR

FORMAL EQUIVALENCE TEST

This is the most work-intensive part of an MCY project, but also what makes MCY special. To know whether the testbench under test *should* return PASS or FAIL, we will set up a formal property check that can conclusively determine whether a mutation can affect the output of the module in a relevant way.

The advantage of using formal methods is that they will exhaustively explore all possible input combinations, which is prohibitive for a simulation testbench for most non-trivial designs due to combinatorial explosion. But the MCY approach is also less difficult than outright formally verifying the design, as it is generally easier to describe whether a change to the output is "important" than to describe the correct behaviour directly.

Unlike in the previous test where we exported the mutated module with the same interface as the original module so we could seamlessly replace it in the testbench, here we will use the `-c` option to get a module where we can enable or disable the mutation at will based on an input signal `mutsel`. We will also export to ILANG format instead of Verilog since SBY understands it.

Create a file `test_eq.sh` and add the following script:

```
#!/bin/bash

exec 2>&1
set -ex

bash $SCRIPTS/create_mutated.sh -c -o mutated.il
```

Next, we will create a miter circuit that compares the original and the mutated module. Create a file named `test_eq.sv` and enter the following code:

```
module miter (
        input [63:0] ref_din_data,
        input [63:0] uut_din_data,
        input [ 2:0] din_func
);
        wire [63:0] ref_dout_data;
        wire [63:0] uut_dout_data;

        bitcnt ref (
                .mutsel    (1'b 0),
                .din_data  (ref_din_data),
                .din_func  (din_func),
                .dout_data (ref_dout_data)
        );

        bitcnt uut (
```

(continues on next page)

(continued from previous page)

```
                .mutsel    (1'b 1),
                .din_data  (uut_din_data),
                .din_func  (din_func),
                .dout_data (uut_dout_data)
        );

endmodule
```

This instantiates the `bitcnt` module twice, once with the mutation disabled (`ref`) and once with the mutation enabled (`uut`). Next, we will add `assert` and `assume` statements that express under which conditions we expect which outputs to be unmodified.

The `bitcnt` module has multiple modes of operation selected by the input `din_func`. The LSB `din_func[0]` selects between 32-bit and 64-bit operand mode, and the MSBs `din_func[2:1]` choose between three counting modes, count leading zeros (CLZ), count trailing zeros (CTZ), or popcount (CNT). The fourth option, `din_func[2:1]==2'b11` is not a valid operation.

The goal is to be as precise as possible about the conditions under which we expect the same output. Therefore we will never check anything in the case of the unused opcode `din_func[2:1] == 2'b11`. We will also disambiguate between the 32 and 64-bit modes and allow the upper input and output bits of `uut` and `ref` to not be identical in 32-bit mode.

At the end of the miter module (before `endmodule`), insert the following code:

```
always @* begin
        casez (din_func)
                3'b11z: begin
                        // unused opcode: don't check anything
                end
                3'bzz1: begin
                        // 32-bit opcodes, only constrain lower 32 bits and only check␣
↪lower 32 bits
                        assume (ref_din_data[31:0] == uut_din_data[31:0]);
                        assert (ref_dout_data[31:0] == uut_dout_data[31:0]);
                end
                3'bzz0: begin
                        // 64-bit opcodes, constrain all 64 input bits and check all 64␣
↪output bits
                        assume (ref_din_data == uut_din_data);
                        assert (ref_dout_data == uut_dout_data);
                end
        endcase
end
```

We will use SBY to check these formal properties. Create the file `test_eq.sby` and enter the following configuration:

```
[options]
mode bmc
depth 1
expect pass,fail

[engines]
smtbmc yices
```

(continues on next page)

(continued from previous page)

```
[script]
read_verilog -sv test_eq.sv
read_ilang mutated.il
prep -top miter
fmcombine miter ref uut
flatten
opt -fast

[files]
test_eq.sv
mutated.il
```

You can consult the SBY documentation for detailed information about how to set up an `sby` project. Points of note here are:

- The `bitcnt` module is combinatorial, so we can use a bounded model check with a single step.
- The additional steps `fmcombine`, `flatten` and `opt` in the script section are not mandatory but increase the speed of the check.
- All files used are assumed to be present in the directory in which the test is run.

You can test your `sby` setup in the `tasks/test` directory with the already created `input.txt` as follows:

```
cd tasks/test
ln -s ../../test_eq.sv ../../test_eq.sby .
bash ../../test_eq.sh
sby -f test_eq.sby
```

As we are once again testing the "do nothing" mutation, this should return `PASS`. If it works correctly, we can complete the script for this test to run `sby` and extract the return value. Append the following to `test_eq.sh`:

```
ln -fs ../../test_eq.sv ../../test_eq.sby .

sby -f test_eq.sby
gawk "{ print 1, \$1; }" test_eq/status >> output.txt

exit 0
```

You can check once more that running `bash ../../test_eq.sh` inside `tasks/test` works correctly and writes `1 PASS` to `output.txt`. Note that the script appends data to this file and an identical line might already exist from previous runs, so verify that a new line is added with the execution.

Finally, set up the configuration for this test at the end of `config.mcy`:

```
[test test_eq]
expect PASS FAIL
run bash $PRJDIR/test_eq.sh
```

CHAPTER

FIVE

TAGGING LOGIC

Now that we have set up the two tests, we need to tell MCY how we want to analyze the results. With two tests, there are only four possible outcomes, which we can each assign a tag:

- both tests fail: the testbench accurately detects the problem, i.e. the mutation is COVERED.
- the simulation testbench passes but the equivalence test fails: the testbench does not find the problem, i.e. the mutation is UNCOVERED.
- the simulation testbench passes and the equivalence test passes: the mutation does not introduce a relevant change to the functionality of the module (NOCHANGE).
- the simulation testbench fails but the equivalence test passes: the equivalence test must not have been set up correctly, and there is a gap between formal description and expected behaviour (EQGAP).

Declare these four tags in the `[options]` section:

```
[options]
size 10
tags COVERED UNCOVERED NOCHANGE EQGAP
```

Then, under the `[logic]` section, describe how to tag the tests:

```
sim_okay = result("test_sim") == "PASS"
eq_okay = result("test_eq") == "PASS"

if sim_okay and not eq_okay:
    tag("UNCOVERED")
elif not sim_okay and not eq_okay:
    tag("COVERED")
elif sim_okay and eq_okay:
    tag("NOCHANGE")
elif not tb_okay and eq_okay:
    tag("EQGAP")
else:
    assert 0
```

This section essentially defines a python function, and can use the predefined functions `result("<name>")` (where `<name>` is a test defined in a `[test <name>]` section) and `tag("<name>")` (for any tag defined under `tags` in the `[options]` section). A single mutation can be tagged with multiple tags, or with no tags at all.

When you have multiple tests of differing length, you can use lazy evaluation to run tests conditionally. For a given mutation, a test is only executed when the `[logic]` section calls `result()`. (An example of this is given in the bonus section at the end of this tutorial.)

Finally, fill in the `[report]` section as follows:

```
[report]
if tags("EQGAP"):
    print("Found %d mutations exposing a formal gap!" % tags("EQGAP"))
if tags("COVERED")+tags("UNCOVERED"):
    print("Coverage: %.2f%%" % (100.0*tags("COVERED")/(tags("COVERED")+tags("UNCOVERED
↪")))))
```

This is again a section that defines a python function. Here, the function `tags("<name>")` can be used to obtain the number of mutations tagged with a given tag. If there is a formal gap, this is highly problematic so it will be reported first. Secondly, we print a coverage metric calculated as the percent of covered mutations out of all mutations that induce a relevant design change, i.e. both those tagged as covered and as uncovered.

CHAPTER

SIX

RUNNING MCY

Now the MCY project is fully set up. Delete the temporary folders `database` and `tasks` we created for testing by running:

```
mcy purge
```

Then, execute MCY:

```
mcy init
mcy run
```

As there are only a few tests requested initially, this should complete quickly. Running in sequential mode (without `-j` argument) makes it more obvious which test is the cause in case of error.

If this initial test run completes successfully and prints a coverage metric, you can increase the number of mutations at the beginning of `config.mcy`:

```
[options]
size 1000
```

This time, the tests will take longer to run, so enable parallel runs (replace `$(nproc)` with the number of cores to use):

```
mcy reset
mcy run -j$(nproc)
```

`reset` will keep the existing results for the previously tested mutations but add more mutations to reach the new requested size.

While the tests are being run, in a second terminal, you can run (in the base project directory where your `config.mcy` is located)

```
mcy dash
```

and open the provided address in your browser to follow progress in the dashboard. This can be especially of interest when running tests on a remote server.

Once the tests complete, you can use:

```
mcy gui
```

to explore visually the hotspots in your code where coverage gaps exist. This is currently hardcoded to use the tag names “COVERED” and “UNCOVERED”.

A similar, command-line-only view is produced by:

```
mcy source bitcnt.v
```

Positive numbers in the left-hand column indicate mutations tagged as COVERED, negative numbers indicate UNCOVERED.

You can try to improve the testbench in `bitcnt_tb.v` to achieve better coverage. After modifying this file, don't forget to invalidate old results by running:

```
mcy purge
```

As mutations are generated randomly, the better your coverage, the larger the size required to find uncovered cases. If you reach 100%, try increasing the size further.

CHAPTER

SEVEN

BONUS: INTEGRATING A SECOND TEST

Often, you will have a whole collection of tests of differing scope and strictness. These can all be integrated into a single MCY project to obtain a coverage metric for the test suite as a whole. In this section we will add a second, longer-running but more thorough testbench to increase the coverage metric.

`test_fm` is a formal testbench that fully verifies that the module fulfils a formal definition of the desired behaviour. Because it significantly increases the runtime of the example, `test_fm` is disabled by default in the `bitcnt` example. It can be enabled or disabled by setting the variable `use_formal` defined in `config.mcy`.

For the purposes of this tutorial, the files `test_fm.sv` and `test_fm.sby` represent a second pre-existing testbench, just like `bitcount_tb.v`. Therefore, simply copy them to your project directory from the `bitcnt` example directory:

```
cp <mcy source dir>/examples/bitcnt/test_fm.{sv,sby} .
```

If you are curious how the formal verification is implemented, you may take a look at the contents. In essence, for each opcode, it asserts that the output conforms to an inductively defined function. For example, for the popcount operation, if `din_data_b` has exactly one more bit set than `din_data_a`, then the count `dout_data_b` should be one higher than `dout_data_a`. This definition is deliberately very different from the implementation of the module, to avoid the common situation where a person writing the same logic twice will make the same errors both times. However, because the `bitcnt` module is so simple, trying to find a different way of expressing it results in a rather more convoluted description than one would usually find in a practical example.

Next, we will create the script to run this test on a mutated design. Create a file named `test_fm.sh` in your project directory with the following contents:

```
#!/bin/bash

exec 2>&1
set -ex

bash $SCRIPTS/create_mutated.sh -o mutated.il

ln -s ../../test_fm.sv ../../test_fm.sby .
sby -f test_fm.sby

gawk "{ print 1, \$1; }" test_fm/status >> output.txt

exit 0
```

Since we are using SBY for this test as well, the script overall resembles `test_eq.sh`. The main difference is that we do not pass `-c` to `create_mutated.sh`, since we need a mutated replacement module with the same interface as the original `bitcnt` module to substitute in the testbench.

As before, we will need the `database/` and `tasks/` directories for a trial run, but this time we can use the existing MCY project to create them.

If the file `database/design.il` does not exist, run `mcy init` to create it.

Next, run `mcy task -k test_sim 1`. Take note of the task uuid printed.

Enter the directory `tasks/${uuid}` created by this command and run `bash ../../test_fm.sh` to check that the test functions correctly (it should return PASS, because task 1 is always `mutate -mode none` which introduces no mutation).

If it works as expected, we can add this test to the MCY configuration. In `config.mcy`, under the section `[options]` reduce the size again while we work and add a new tag `FMONLY`:

```
[options]
size 10
tags COVERED UNCOVERED NOCHANGE EQGAP FMONLY
```

At the bottom of the file, add a new section for the new test:

```
[test test_fm]
expect PASS FAIL
run bash $PRJDIR/test_fm.sh
```

Finally, we will adjust the `[logic]` section to use this new test. First, define the variable `use_formal` so we can turn on and off this expensive test at will:

```
[logic]
use_formal = True
```

Second, after the two original tests are run, but before the tags are applied, insert a new piece of code:

```
tb_okay = (result("test_sim") == "PASS")
eq_okay = (result("test_eq") == "PASS")

if tb_okay and use_formal:
    tb_okay = (result("test_fm") == "PASS")
    if not tb_okay:
        tag("FMONLY")

if tb_okay and not eq_okay:
    tag("UNCOVERED")
elif ...
```

This will run `test_fm` only in the case where `use_formal` is enabled and `tb_okay` is true, i.e. the simulation testbench did not identify any problem with the module. This means that this long-running test will only be executed for a small portion of the mutations.

As the variable `tb_okay` is potentially modified in this `if` before the original tagging logic runs, the `COVERED` tag is now applied to any mutation that was caught by either the simulation or the formal verification testbench. Mutations for which only the formal test was able to detect a problem are tagged with `FMONLY` so that we can trace which tests cover which mutations.

Test that this new configuration works correctly:

```
mcy purge
mcy init
mcy run
```

Depending on your randomly generated mutations, you may have some mutations tagged as `FMONLY` in your initial set of 10. Check if the following line appears in `mcy status`:

```
Tagged 1 mutations as "FMONLY".
```

If you wish, you can generate new mutations by re-running the above commands, or by increasing the number of mutations.

If everything is working correctly, you can return the mutation set size to its original value.

```
[options]
size 1000
```

Running MCY will now require significantly more time, so don't forget to enable parallelism:

```
mcy reset
mcy run -j$(nproc)
```

This time, you should achieve 100% coverage, as the formal testbench comprehensively checks whether the output is correct for any possible combination of inputs.