Project-Part 4

EE271

September 2020

1 Catching Difficult Bugs with A-QED

By this point, you should (hopefully) have a working rasterizer module that passes all the test vectors. However, as mentioned in class, test vectors are difficult to create. As a result, passing test vectors does not guarantee a bug-free design. In fact, here's a surprise: the rasterizer starter code given to you had been injected with bug(s). And in spite of these bugs, all the test vectors will still pass.

In this part of the project, we will explore a new verification technique: A-QED. As mentioned in class, A-QED outperforms conventional verification methods in catching bugs that are triggered by a sequence of inputs, with the **Functional Consistency Check**. Using A-QED, let's see if we can catch these injected bugs.

NOTE: It will be easier to use your design from Part 2, not the optimized version from Part 3.

1.1 Designing the A-QED module

First, we need to create an A-QED module and connect it to the rasterizer to perform our verification (Fig. 1). This A-QED module will drive inputs to the accelerator and monitor the outputs. In order to perform the **Functional Consistency Check**, two identical inputs are sent: **original** and **duplicate**, with an arbitrary number of inputs between them. Output sub-sequences corresponding to the original and the duplicate inputs are then compared, and since the inputs are identical, the outputs must also be identical.

Recall the first part of interfacing the A-QED module with the accelerator is to identify the interface signals. The interfacing signals are **halt_RnnnnL** (input_ready) and **validTri_R10H** (input_valid). In addition, we need an output_valid signal, which requires us to slightly modify our design.

Take a look at Figure 2. We will use the halt signal to create the output_valid signal. Note that there is a four stage pipeline from the test_iterator where the halt signal is created, to where the outputs are produced. Therefore, we will need to create a 5 cycle delayed halt signal called halt_d5.

Note that for an input, outputs can be produced over multiple cycles. Let us call these outputs corresponding to one input as an output sub-sequence. We need to compare the entire output sub-sequence generated from the original input to that generated from the duplicate. Note that a sub-sequence is produced as long as the halt_d5 is low. So, the completion of a sub-sequence is marked by the transition of halt_d5 from 0 to 1.

Now that we have all our interface signals, complete **formal/aqed.sv**, using the comments as a guide.

1.2 Catching Bugs

In this part, we will be using a commercial formal verification tool, Cadence JasperGold.

To run A-QED verification,

```
$ cd formal
$ ./run-AQED.sh
```

You should now see the JasperGold GUI open and run BMC on the rasterizer combined with the A-QED module. If there is something wrong in the your code, JasperGold throws an error in the console also indicating the line number. If you click on the line number, it takes you to the RTL line where there is a problem. If it can find a counter-example, in the *property table*, you will see a red cross beside the Functional Consistency Check assertion **aqed.assert_qed_match**. Along side, the **visualize window** will pop up from where you can inspect the waveform traces to pin-point the problem. We have already provided a list of signals that you can load into the **visualize window**. To do this, in the **visualize window** go to

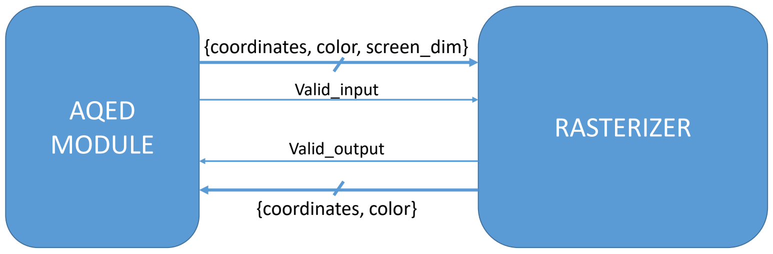

Figure 1: A-QED Setup

```
$ File -> Load Signal List.. -> debug.sig
```

To find the cause of the counter-example, you will need to figure out the original and the duplicate outputs and find why there is a mismatch between them. You can right-click on a signal at a particular cycle and press *why*. This will take you to its driver signals in the RTL and display the particular values in the driver signals because of which the current signal was assigned that value. You can recursively use the *why* feature on the driver signals and figure out the root for the inconsistency between them thus figuring out the root cause for the inconsistency between the original and duplicate outputs.

Keep an eye out for spurious counter-examples. From the trace generated, try to understand what the cause of the counter-example is. If the cause maps back to the A-QED module then the counter-example is not a true bug, but rather an error in the A-QED module.

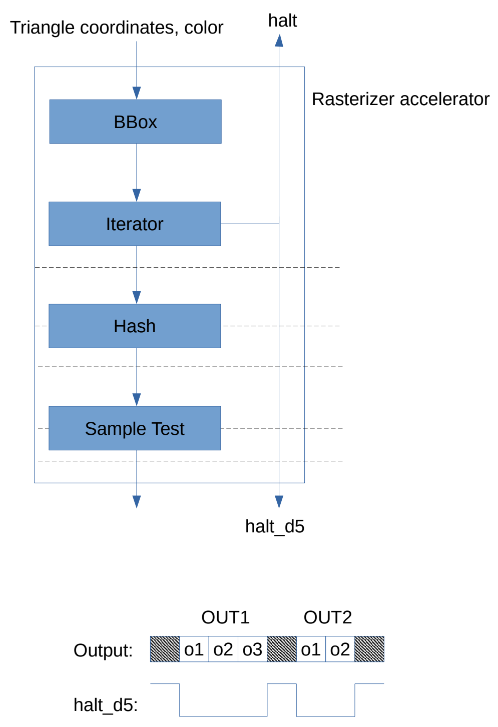

Figure 2: New signal halt_d5 created to mark the start and stop of an output sub-sequence. If two inputs IN1 and IN2 are send by the host, the corresponding outputs OUT1 and OUT2 may spread across multiple cycles. halt_d5 marks the start and stop of these multi-cycle output sub-sequences.

Otherwise, you have found a true bug! For each bug found, explain why the bug is creating a mismatch between the output sub-sequences corresponding to the original and the duplicate inputs although they are equal, and fix the bug. Continue rerunning A-QED verification and fixing bugs until you no longer find any counter-examples. Allow an hour or more for A-QED to run on the design that you think is free of bugs.

1.3 Write-Up

For the write-up, please include the following:

- A brief description of each bug found
- Explain why the conventional simulation and formal verification missed these bugs.

1.4 Submission

Clean up the folder first:

```
$ make cleanall
```

And then zip it:

```
$ zip -r assignment4.zip /path/to/rasterizer/folder
```

Submit to Gradescope by **Friday, November 20, 11:59PM PST**.