- Remember to register your attendance using the UoB Check-In app. Either
  1. download, install, and use the native app[a] available for Android and iOS, or
  2. directly use the web-based app available at

     https://check-in.bristol.ac.uk

  noting the latter is also linked to via the `Attendance` menu item on the left-hand side of the Blackboard-based unit web-site.
- The hardware *and* software resources located in the MVB Linux lab(s). (e.g., MVB-1.15 or MVB-2.11) are managed by the Faculty IT Support Team, a subset of IT Services. If you encounter a problem (e.g., a workstation that fails to boot, an error when you try to use some software, or you just cannot log into your account), they can help: you can contact them, to report then resolve said problem, via

  https://www.bristol.ac.uk/it-support

- The lab. worksheet is written *assuming* you work in the lab. using UoB-managed and thus *supported* equipment. If you need or prefer to use your own equipment, however, various *unsupported*[b] alternatives available: for example, *you* could 1) manually install any software dependencies yourself, *or* 2) use the unit-specific Vagrant[c] box by following instructions at

  https://cs-uob.github.io/COMS10015/vm

- The questions are roughly classified as either C (for core questions, that *should* be attempted within the lab. slot), A (for additional questions, that *could* be attempted within the lab. slot), or R (for revision questions). Keep in mind that we only *expect* you to attempt the C-class questions: the other classes are provided *purely* for your benefit and/or interest, so there is no problem with nor penalty for totally ignoring them.
- There is an associated set of solutions is available, at least for the C-class questions. These solutions are there for you to learn from (e.g., to provide an explanation or hint, or illustrate a range of different solutions and/or trade-offs), rather than (purely) to judge your solution against; they often present *a* solution vs. *the* solution, meaning there might be many valid approaches to and solutions for a question.
- Keep in mind that various mechanisms exist to get support with and/or feedback on your work; these include both in-person (e.g., the lab. slot itself) *and* online (e.g., the unit forum, accessible via the unit web-site) instances.

[a] https://www.bristol.ac.uk/students/support/it/software-and-online-resources/registering-attendance
[b] The implication here is that such alternatives are provided in a best-effort attempt to help you: they are experimental, and so *no* guarantees about nor support for their use will be offered.
[c] https://www.vagrantup.com

COMS10015 lab. worksheet #5 + #7

During the period of time aligned with this lab. worksheet, there is an active (or open) coursework assignment for the unit. You could address this fact by dividing your time between them. However, our (strong) suggestion is to view the former as of secondary importance (or optional, basically), and instead focus on the latter: since it is credit bearing, the coursework assignment should be viewed as of primary importance. Put another way, focus exclusively on completing the latter before you invest any time at all in the former.

Before you start work, download (and, if need be, unarchive[a]) the file

`https://assets.phoo.org/COMS10015_2024_TB-4/csdsp/sheet/lab-07_q.tar.gz`

somewhere secure[b] in your file system; from here on, we assume `${ARCHIVE}` denotes a path to the resulting, unarchived content. The archive content is intended to act as a starting point for your work, and will be referred to in what follows.

[a] For example, you could 1) use `tar`, e.g., by issuing the command `tar xvfz lab-07_q.tar.gz` in a terminal window, 2) use `ark` directly: use the `Activities` desktop menu item, search for and execute `ark`, use the `Archive→Open` menu item to open `lab-07_q.tar.gz`, then extract the contents via the `Extract` button, or 3) use `ark` indirectly: use the `Activities` desktop menu item, search for and execute `dolphin`, right-click on `lab-07_q.tar.gz`, select `Open with`, select `ark`, then extract the contents via the `Extract` button.

[b] For example, the `Private` sub-directory within your home directory (which, by default, cannot be read by another user).

§1. C-class, or core questions

▷ **Q1[C].** This and subsequent lab. worksheets make use[1] of Icarus Verilog (referred to using the short-hand Icarus) and GTKWave, tools which support synthesis and simulation of Verilog models expressed as source code: doing so enables hands-on exploration of associated concepts, and thus aims to enhance your understanding based on the lecture slot(s) alone. Note that although you *could* download and install Icarus and GTKWave yourself, it is not *necessary* to do so because they have been pre-installed, e.g., on the lab. workstations.

A brief introduction. The goal[2] of *this* question is to explain the high-level features in and workflow for using these tools, offering experience that will allow you to engage with tasks and challenges presented in *later* questions. Before you start:

a update your `${PATH}`[3] environment variable by executing

```
export PATH="${PATH}:/opt/iverilog/12.0/bin"
```

b check said update worked correctly by executing

```
which iverilog
which gtkwave
```

noting that any reported error (e.g., `no iverilog in ...` or similar) suggests it did not: ask for help!

Imagine you have 1) `X.v`, which implements a module `X`, plus 2) `X_test.v`, which implements a module `X_test`, where the latter acts as a test stimulus for the former. Although all of the steps related to use of `X.v` and `X_test.v` could be performed manually, the `Makefile` provided represents an automated build system which implies less effort *and* less chance of error. Based on the use of `Makefile`, an edit-compile-execute style design cycle can be roughly summerised as follows:

a Edit the Verilog source code file `X.v`.

b Execute

```
make clean
```

to clean (i.e., remove) residual files stemming from previous compilation or simulation steps.

[1] See, e.g., `https://github.com/steveicarus/iverilog` and `https://github.com/gtkwave/gtkwave` noting that although we focus specifically on Icarus version 11.0 and GTKWave version 3.3.104, use of *other* versions is plausible modulo any minor incompatibilities.

[2] The high-level focus implies that a *non*-goal is therefore an in-depth tutorial-style introduction. Rather, there is an emphasis on *you* to, e.g., explore and use the wider documentation as and when need be: see, e.g., `https://steveicarus.github.io/iverilog` and `https://gtkwave.sourceforge.net/gtkwave.pdf`

[3] `http://en.wikipedia.org/wiki/PATH_(variable)`

c Execute

```
make X_test.vvp
```

to compile the design using `iverilog`: doing so combines the Verilog source code file `X.v` with the associated test stimulus `X_test.v` to produce the executable `X_test.vvp`.

d Execute

```
make X_test.vcd
```

to simulate the design using `vvp`: doing so produces 1) some machine-readable output via a Value Change Dump (VCD)[4] file `X_test.vcd`, plus 2) some human-readable output via the terminal.

Not all test stimuli are fully-automatic; some require arguments (or parameters) to control them. In such a case, `Makefile` uses the `ARGS` environment variable to capture arguments it then passes to `vvp`. For example, imagine `X_test` requires three 1-bit arguments called `x`, `y`, and `z`: instead of the above, one could instead execute

```
make ARGS="+x=0 +y=1 +z=0" X_test.vcd
```

to set `x` = 0, `y` = 1, and `z` = 0.

e Execute

```
gtkwave X_test.vcd
```

to visualise the resulting VCD file using `gtkwave`.

The files `example.v` and `example_test.v` in the archive provided model a half-adder[5] component; they are reproduced in Figure 1 and Figure 2 respectively. Using them by setting `X` = `example`, consider a case where we follow the workflow above and produce `example_test.vcd` as output: by then executing GTKWave to visualise this output, you *should* eventually see something similar to Figure 3 which has been annotated to highlight several major features:

- The signals pane represents a so-called Signal Search Tree (SST): the top section is a tree (i.e., the hierarchy) of module instances, and the bottom section is a list of signals available within the selected component instance.
- The waveform signals pane is a list of signals, which were identified using the SST then added in order to visualise them.
- The waveform pane acts as a visualisation of each waveform signal: it shows 1) visually how the signal changes over a period of time (each shown as a horizontal line, or "bubble" for wire vectors), and 2) numerically what value the signal has at a given point in time dictated by a marker (shown as a vertical line). Note that the horizontal and vertical scrollbars allow the viewport to be moved around, which changes the period and subset of signals visualised.
- The button pane contains a number of short-cuts for common operations: some examples include
  - zoom the waveform in and out, i.e., decrease or increase the visualised period of time,
  - move to the start or end of the waveform,
  - move to the previous or next edge for a selected waveform signal.

Figure 4 shows two approaches to visualising the behaviour of the modelled half-adder. The left-hand side of Figure 4 offers an external perspective, i.e., from the perspective of the top-level instance of `example_test`. Figure 4a shows that by clicking on the module instance within the SST populates the available signal list, then Figure 4e shows that appending all available signals to the waveform acts to visualise how they change between 0s and 50s: setting the marker to 25s shows that if `x` = 0 and `y` = 1 then `co` = 0 and `s` = 1 as expected. The right-hand side of Figure 4 offers an internal perspective, i.e., from the perspective of the instance of `example` within the top-level instance of `example_test`. The end result in Figure 4e is the same as Figure 4f; the difference is that instead of using the external signals (i.e., `t_co`, `t_s`, `t_x`, and `t_y`) provided to the instance of `example`, we use the internal signals (i.e., `co`, `s`, `x`, and `y`) which are accessible *within* the instance of `example`.

[4] https://en.wikipedia.org/wiki/Value_change_dump
[5] https://en.wikipedia.org/wiki/Adder_(electronics)

Some initial, exploratory tasks.

a To acclimatise yourself with the workflow involved, reproduce the example above: use the marker to verify that the output produce by the instance of `example` is correct for each valid input combination.

b Complete the implementation of modules called `add_1bit` and `add_1bit_test`, which model a 1-bit full-adder: doing so essentially means extending `example` and `example_test` to support the additional `ci` (or carry-in) input.

c Produce implementations of modules called `add_4bit` and `add_4bit_test` which model a 4-bit ripple-carry adder: doing so essentially means connecting together 4 instances of `add_1bit`.

▷ **Q2[C].** This question is a follow-on from Question 1, focusing on combinatorial logic design using Verilog.

a The archive provided includes an incomplete, skeleton implementation of a module called `ads_4bit`: it models an enhanced *unsigned* 4-bit adder, in the sense that

$$\texttt{r} = \begin{cases} \texttt{x} + \texttt{y} + \texttt{ci} & \text{if } \texttt{mode} = 0 \\ \texttt{x} - \texttt{y} - \texttt{ci} & \text{if } \texttt{mode} = 1 \end{cases}$$

Research and then implement a design for `ads_4bit` so it can compute `r`, aiming to optimise for low-area (i.e., minimise the total number of logic gates required). Then, as a second step, enable it to compute

$$\texttt{of} = \begin{cases} 1 & \text{if an overflow condition does occur during computation of } \texttt{r} \\ 0 & \text{if an overflow condition does not occur during computation of } \texttt{r} \end{cases}$$

$$\texttt{co} = \begin{cases} 1 & \text{if a carry condition does occur during computation of } \texttt{r} \\ 0 & \text{if a carry condition does not occur during computation of } \texttt{r} \end{cases}$$

noting that `of` and `co` are only relevant if `x`, `y`, and `r` are interpreted as signed or unsigned integers respectively.

b The archive provided includes an incomplete, skeleton implementation of a module called `cmp_4bit`: it models an *unsigned* 4-bit comparator, in the sense that

$$\texttt{lth} = \begin{cases} 1 & \text{if } \texttt{x} < \texttt{y} \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{equ} = \begin{cases} 1 & \text{if } \texttt{x} = \texttt{y} \\ 0 & \text{otherwise} \end{cases}$$

$$\texttt{gth} = \begin{cases} 1 & \text{if } \texttt{x} > \texttt{y} \\ 0 & \text{otherwise} \end{cases}$$

Research and then implement a design for `cmp_4bit` so it can compute `lth`, `equ`, and `gth`, aiming to optimise for low-area (i.e., minimise the total number of logic gates required).

▷ **Q3[C].** This question is a follow-on from Question 1, focusing on sequential logic design using Verilog. The archive provided includes an incomplete, skeleton implementation of modules called `ctr_4bit_latch` and `ctr_4bit_fflop`: they model a 4-bit counter, implemented using latches and flip-flops respectively.

a Implement the modules and their corresponding test stimuli `ctr_4bit_latch_test` and `ctr_4bit_fflop_test`, so the latter initially reset the counter before allowing the clock signal(s) to drive updates of it: at this stage, you can ignore the `mode` input.

b Update your implementation to now consider the `mode` input: the idea is that, per lab. worksheet #4, if `mode` = 0 the counter exhibits cyclic behaviour whereas `mode` = 1 the counter exhibits saturating behaviour.

```
8  module example( output wire co,
9                  output wire  s,
10
11                 input  wire  x,
12                 input  wire  y );
13
14   xor t0(  s, x, y );
15   and t1( co, x, y );
16
17 endmodule
```

Figure 1: *example.v.*

```
8  module example_test();
9
10   wire t_co;
11   wire t_s;
12   reg  t_x;
13   reg  t_y;
14
15   example t( .co( t_co ), .s( t_s ), .x( t_x ), .y( t_y ) );
16
17   initial begin
18         $dumpfile( "example_test.vcd" );
19         $dumplimit( 10485760 );
20         $dumpvars;
21
22         $dumpon;
23
24     #10 t_x = 1'b0; t_y = 1'b0;
25     #10 t_x = 1'b0; t_y = 1'b1;
26     #10 t_x = 1'b1; t_y = 1'b0;
27     #10 t_x = 1'b1; t_y = 1'b1;
28
29     #10 $dumpoff;
30         $finish;
31   end
32
33 endmodule
```

Figure 2: *example_test.v.*

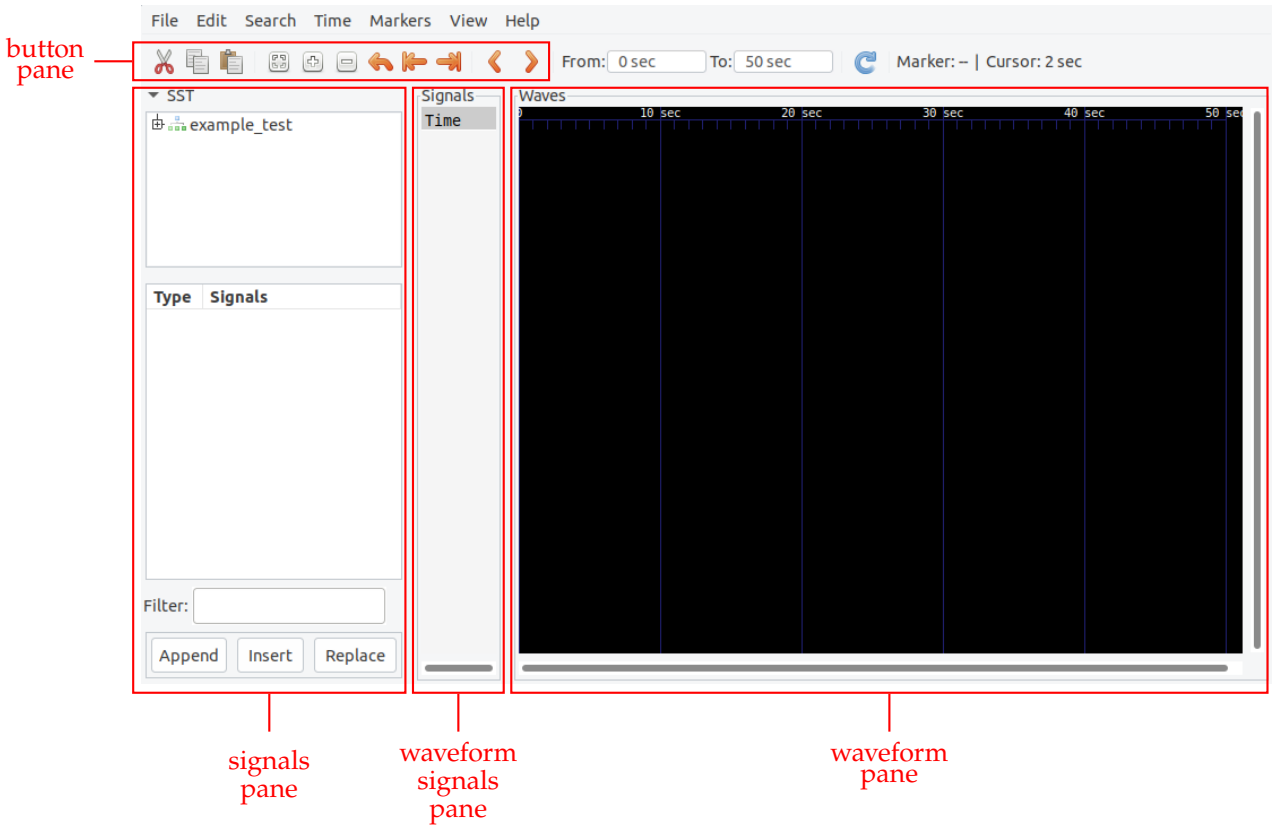
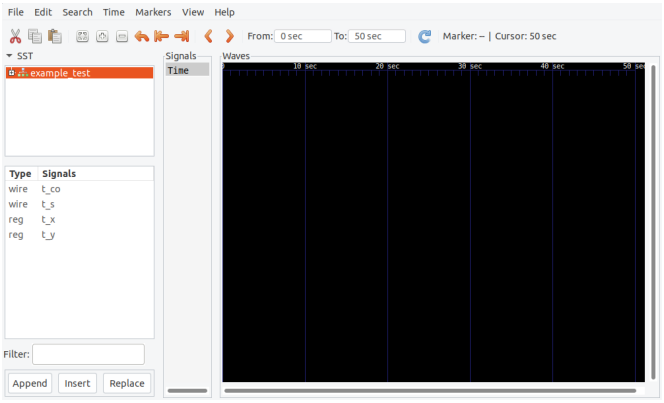
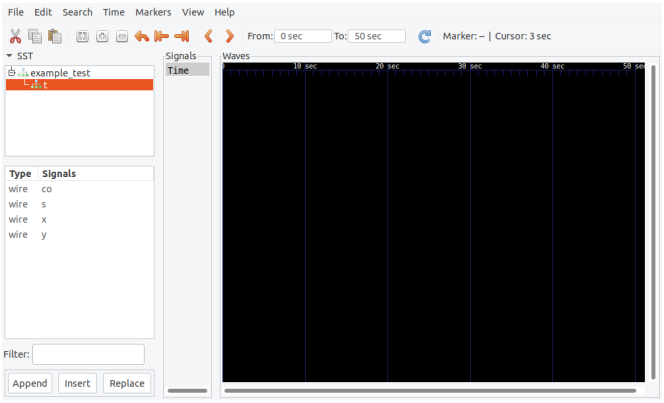


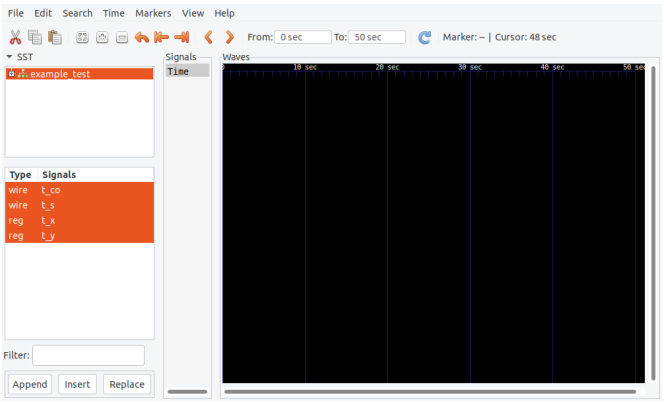
Figure 3: *The GTKWave user interface, annotated to illustrate major features.*



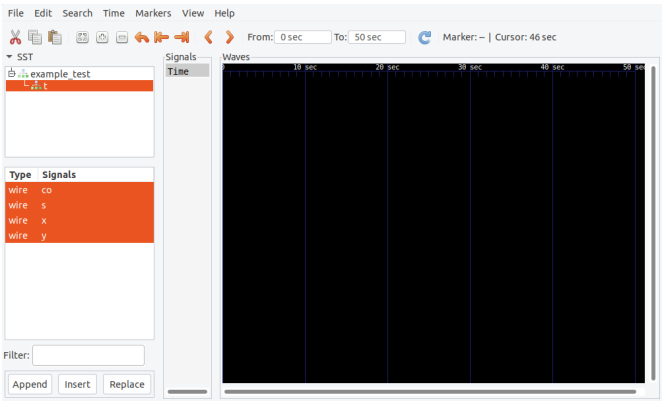
(a) *Select* `example_test` *instance (i.e., the top-level instance of* `example_test`*).*



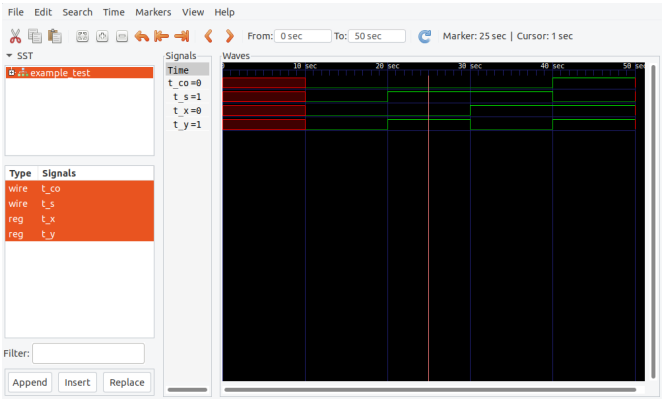
(b) *Select* `example_test.t` *instance (i.e., the instance of* `example` *within the top-level instance of* `example_test`*).*



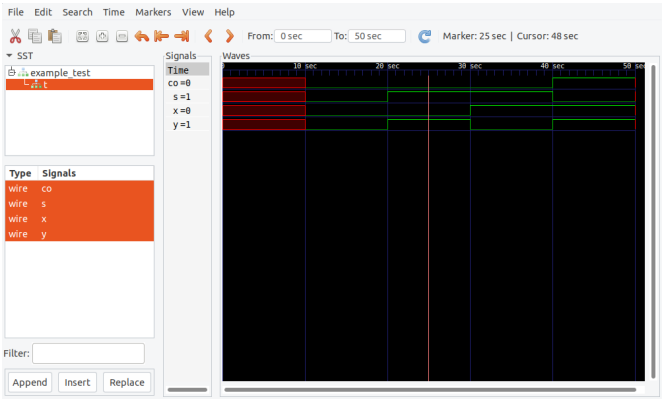
(c) *Select all available signals.*



(d) *Select all available signals.*



(e) *Append all selected signals (using the* Append *button); set marker to* 25s.



(f) *Append all selected signals (using the* Append *button); set marker to* 25s.

Figure 4: *An example using GTKWave: external (left) and internal (right) perspectives on behaviour of an instance of the* `example` *module.*

§2. R-class, or revision questions

▷ **Q4[R].** There is a set of questions available at

https://assets.phoo.org/COMS10015_2024_TB-4/csdsp/sheet/misc-revision_q.pdf

Using pencil-and-paper, each asks you to solve a problem relating to Boolean algebra. There are too many for the lab. session(s) alone, but, in the longer term, the idea is simple: attempt to answer the questions, applying theory covered in the lecture(s) to do so, as a means of revising and thereby *ensuring* you understand the material.

§3. A-class, or additional questions

▷ **Q5[A].** In Question 1, you already produced an implementation of `add_4bit` and `add_4bit_test` which model a 4-bit ripple-carry adder. Treating this as a starting point, it can be useful to consider further improvements which focus on the implementation (and the source code which describes it) rather than the design per se: rather than correctness, these impact metrics such as generality and maintainability for example.

a `add_1bit` has three inputs, namely a 1-bit `ci`, a 1-bit `x`, and a 1-bit `y`, meaning there are $2^1 \cdot 2^1 \cdot 2^1 = 2 \cdot 2 \cdot 2 = 8$ possible input combinations; `add_1bit_test` capitalises on this fact using an exhaustive (or brute force) approach to testing. The same approach becomes less feasible for `add_4bit`, which has a 1-bit `ci`, a 4-bit `x`, and a 4-bit `y`, and therefore $2^1 \cdot 2^4 \cdot 2^4 = 2 \cdot 16 \cdot 16 = 512$ possible input combinations.

To address this challenge, one can imagine an approach that involves (at least) the following steps:

i Use automated (versus, e.g., manual, hard-coded) iteration through input combinations; this approach can be realised, e.g., by employing the Verilog `for` statement.

ii Use automated (versus, e.g., manual, by inspection) identification of input combinations which yield an incorrect output; this approach can be realised, e.g., by employing an oracle[6] based on the Verilog + operator.

Try to explore, then apply these steps in your implementation.

b One could view `add_4bit` as implementing an n-bit ripple-carry adder for the specific case of $n = 4$; allowing the implementation to be more general, i.e., support *any* n, is clearly attractive versus implementing a different module for *each* n.

To address this challenge, one can imagine an approach that involves (at least) the following steps:

i Introduce a module parameter `n`.

ii Use `n` to, e.g., generalise the number of bits in `x`, `y`, and `r`.

iii Use automated generation of (versus, e.g., manual hard-coding of) `add_1bit` instances; this approach can be realised, e.g., by employing the Verilog `generate` statement.

Try to explore, then apply these steps in your implementation.

[6] https://en.wikipedia.org/wiki/Test_oracle