DESIGN VERIFICATION Practical 2: Debugging a Calculator Design

The aim of this practical is for you to explore advanced verification methodologies and to experience the power of automation in simulation-based verification with state-of-the-art industrial verification tools.

This practical requires you to find bugs in the same calculator design as used for Practical 1, but this time you should explore the use of the e language and the Specman Elite verification environment to perform the verification. In particular, this practical offers you the opportunity to practice and develop skills in using an advanced verification methodology including constrained pseudo-random test generation, automatic checking and different forms of coverage.

For this practical, the calc1 calculator design source code is provided. An example .e testbench has been demonstrated during a live session. This testbench implements basic constrained pseudo-random test generation, a simple check for the ADD command and shows how coverage can be collected. The .e code of this testbench is provided to give you a starting point.

Calculator Specification - [same functionality as for Practical 1]

Input/Output Specification

The calculator has 4 commands: add, subtract, shift left and shift right. It can handle (but not process) 4 requests in parallel. All 4 requestors use separate input signals. All requestors have equal priority. Figure 1 shows the input output ports of the calculator.

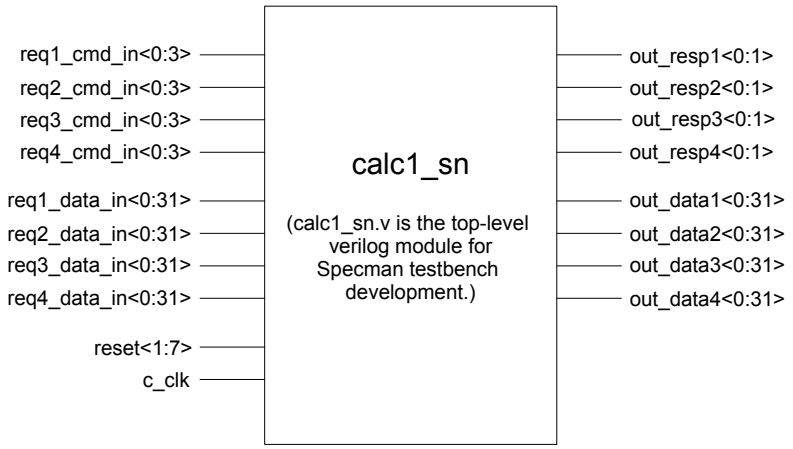

Figure 1: Top-level of Calculator Design for Specman Testbench Development

Input commands are coded as follows:

0 no op
1 add operand_1 and operand_2
2 subtract operand_2 from operand_1
5 shift left operand_1 by operand_2 places
6 shift right operand_1 by operand_2 places

Input data timing: Operand_1 arrives with the command. Operand_2 arrives on the following clock cycle.

Output response lines encode the following:

0 no response
1 successful operation completion
2 invalid command or overflow/underflow error
3 internal error

Output data timing: Valid result data on output lines accompanies the response on successful completion (i.e. both are driven in the same cycle). Figure 2 depicts the input/output timing on a single port of the calculator. The lightening bolt represents "some number of cycles passing". The data accompanies a successful response signal. A second request from the same port is prohibited until the response from the first command is received.

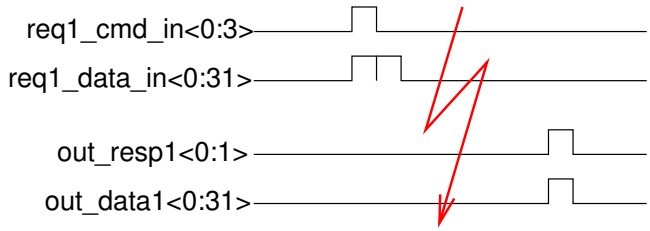

Figure 2: Input/Output Timing of Calculator Design

Clocking: The clock should be toggled when using an event-based simulator.

Priority Specification

The calculator priority logic works on a first come first serve algorithm. It allows for 1 add or subtract at a time and one shift operation at a time.

In other words, if all 4 ports send an add/subtract at once, then all will be processed one after another (not in parallel). None of the requests will be lost or dropped. It just takes additional cycles to process the 2nd, 3rd, and 4th commands.

Specification FAQs clarified:

The inputs are UNSIGNED, hence FFFFFFFF + 00000001 is intended to be an overflow case. Also, subtraction of a larger number from a smaller number should yield an underflow.

Shift operations do not overflow or underflow, they drop the bits as they shift out, and zeros are shifted in.

Suggested Verification Strategy

- Familiarise yourself with the specification. Make sure you understand what the calculator is supposed to do.
- Plan *what* you want to verify and *how* you are going to do this. Your plan serves as the *specification for your testbench*. In particular, *define suitable driving and checking strategies, and coverage metrics*.
- Design your testbench. Aim to distribute the functionality of your testbench into dedicated modules. Keep the tests separate from the verification environment, and make the checkers independent of the tests. Use the example testbench modules as a guide for modularization.

  Overview of example testbench modules:

  - `calc1_sn_env.e` is the top level testbench module which provides the basic structure for the calc1 testbench environment. It imports the other modules and is the place where you can customize Specman.
  - `instruction.e` provides the basic structure for the calc1 design instructions with 3 physical fields and two fields marked so that values are not generated for them. It also provides a type declaration for calc1 operation encoding, and a very basic example response checker for ADD instructions.
  - `driver.e` provides the unit for the calc1 testbench driver. The driver interacts directly with the calc1 design via a number of ports that link the driver to the actual signals in the calc1 design. The driver drives test data (a sequence of instructions) into the design and collects the response from the design. It also invokes the instruction-specific response checker. The sequence of instructions to be driven into the calc1 design is defined in the unit driver_u as a list of instructions.
  - `coverage.e` provides a basic example of coverage collection for the calc1 testbench. It extends the previously defined instruction struct with a declaration of the event `instruction_complete` and a functional coverage metric to cover this event. To trigger coverage collection, the driver struct is extended to emit the `instruction_complete` event at the end of the `collect_response` method.
  - `test_example.e` provides the basic test-specific constraints for the calc1 testbench. Notice how the test constrains the random generation of data for the physical fields in the instruction data structure. It only generates ADD commands with both data items constrained to values within a specified limit. In addition, it constrains the size of the instruction sequence to be driven into the design to a fixed number of instructions for this test.

- Implement your testbench in e and use it to verify the calculator design.
- Use coverage to drive the verification. Make sure you include the coverage specific elements when you invoke `irun`.

  When you run the simulation with a functional coverage model defined in a .e coverage module such as `coverage.e` included into the top-level testbench environment `calc1_sn_env.e` then functional coverage will be collected during simulation. You must close the tools before coverage data is written.

  After simulation you can view the coverage results, including functional coverage, by calling `imc`.

  This opens IMC in a new window.

  Specman coverage files are stored in the same directory structure as the RTL coverage, e.g.

  `cov_work/scope/test/*.ucd`

`cov_work` is the default name for the working directory where all the coverage data goes - it can be renamed via the `irun` option `-covworkdir` if you want. A typical use for this option is to put all the coverage into the same `cov_work` directory when you're running tests in parallel from many different directories.

`scope` is the default name for a concept Cadence refers to as the verification scope. What this does is give you a way to group all your coverage results in a sub-directory that's related to the verification activity, e.g. you may have scopes for module-level, system-level RTL and gate-level netlist, all under the same `cov_work` directory. The `irun` option `-covscope` lets you specify this directory name, relative to `cov_work`. Coverage model (`*.ucm`) files are stored in here and are shared between the tests that use that model.

`test` is the default name for a single test run and is controlled by the `irun` option `-covtest`. If you're running many seeds for the same test it's a good idea to use the `-covtest` option to explicitly name the test directory. Two `*.ucd` data files are stored in each test directory, one for structural coverage from ncsim and one for Specman. When you ask IMC to load a test, you just point to the `cov_work/scope/test` directory and IMC will locate the correct `*.ucd` and `*.ucm` files to load in.

To get cumulative coverage results, you have to use IMC in batch mode, which merges coverage results from different runs. (This may change in future.) To merge:

```
imc -batch
imc> merge -out cov_work/scope/merged/ cov_work/scope/test*
```

You can then load the merged file in GUI mode or generate an HTML report from batch mode via the "report" command.

- Again, be as precise as you can when describing bugs! When you find a bug, try to narrow it down as much as you can. For example, it is more helpful if you tell a designer that: "Subtraction does not work on bits 8 and 9 (counted from the least significant bit upwards)." than if you say: "Subtraction does not work." or "It does not do 100000 - 6 correctly."
- Remember that your testbench needs to be generic; it should not be specific to the version of the calculator you are working with. This means that, even if we were to change the implementation of the calculator, e.g. next week and again the week after, then your testbench should still be effective at finding bugs in any new version of the calculator.

Basic setup to run Specman:

To get access to the EDA tools, please follow the instructions given by IT Services as provided on the unit webpage.

Running Specman together with Incisive Unified Simulator (ISU):

1. Make sure you've got all relevant `.e` files and all relevant `.v` files in one directory. `cd` into this directory.
2. Issue the following command:

   ```
   irun calc1_sn_env.e calc1_sn.v -gui -access rw -coverage all
   -covtest my_code_coverage_results
   -snprerun "config cover -write_model=ucm" -nosncomp &
   ```

   This should bring up SimVision with both the "Specman" as well as the "simulator" tab available from the SimVision Console window.
3. The top level testbench module `calc1_sn_env.e` has been loaded via the command line.
4. To practice working with the tools, follow the instructions in the `SN_DEMO_HOWTO` file that was used for the in class Specman demo session.