Politecnico di Torino

Department of Electronics and Telecommunications

Multidisciplinary Workshop
SoC Verification Strategies

Step 1: Technical Report

Student: s315216@studenti.polito.it

June 16, 2023

Contents

1 Introduction **3**
1.1 The Verification Plan . . . 3
1.2 Directed Testing . . . 3
1.3 Find Bugs Faster . . . 3
1.4 The Hands-On Experience . . . 4

2 The Testbench Architecture **5**
2.1 The Core Components . . . 5
2.2 DUT - Arithmetic-Logic Unit . . . 7
2.2.1 The Testbench Refinement . . . 10
2.3 DUT - Accumulator . . . 11
2.3.1 The Testbench Refinement . . . 14

3 Results **14**
3.1 Arithmetic-Logic Unit . . . 14
3.1.1 Debugging . . . 16
3.2 Accumulator . . . 17

4 Conclusion **19**

List of Listings

2.1 The snippet highlights how type template parameters can improve code reusability, greatly reducing development time. `$cast()` here is an example of dynamic downcasting: `copy()` returns a base class handle that gets cast to point to the child class `T`. 6
2.2 VHDL interface of the Arithmetic-Logic Unit (ALU) under test. `type_op` is an enumerated type defined in a VHDL package, imported into SystemVerilog namespaces when necessary. 7
2.3 Weighted distribution constraint for the ALU input operands. 8
2.4 Comparison between: driving the ALU inputs through the interface clocking block as synchronous signals. Reading the ALU outputs through the interface as an asynchronous signal but synchronized with the same active edge of the clocking block. 8
2.5 Functional coverage specifications for the ALU under test. 9
2.6 Snippet of the golden model for the ALU under test, implemented as a driver callback. 9
2.7 VHDL interface of the accumulator under test. 12
2.8 Snippet of the golden model for the accumulator under test, implemented as a driver callback. 12
2.9 Functional coverage specifications for the accumulator under test. Being it a sequential circuit, there are some additional cases worth reporting to measure how extensively the functionality of the Device Under Test (DUT) was tested. In the cross-coverage specification, the `acc_cp` and `acc_en_n_cp` cover-points would automatically originate 4 bins: here the two bins corresponding to `acc_en_n_cp == 1'b1` are merged because both are associated to the case of the output register of the accumulator being disabled. 13

List of Figures

1 Conceptual representation of the layered testbench environment and its interface to the DUT. The dashed arrows highlight the callback calls; the letter envelopes mark the path followed by transactions around the testbench. 5
2 Visualization of the simulation logs for the ALU under test: 32 bit parallelism, 100 packets. 15
3 Additional run of the ALU simulation: 32 bit parallelism, 1000 packets. Both *funclsr* and *funclsl* are faulty. 16
4 Visualization of the simulation logs for the accumulator under test; 32 bit parallelism, 100 packets. 18

1 Introduction

Verification engineers strive to find bugs as early as possible in the design cycle because the cost of fixing them increases when getting closer to tape-out. The verification process goes in parallel to design creation and the starting point is in both cases the design specifications. Taking for granted that there is always ambiguity in this kind of natural language description, having both the verification engineers and the hardware designers perform the same interpretation adds redundancy in the process, each of them making an independent assessment of what those specifications mean.

At this crossroad, goals diverge; the verification team has to make sure that, whatever the design team comes up with, is indeed compliant with the specifications and simulation is still the workhorse to achieve that.

1.1 The Verification Plan

Although testbenches share a common structure of stimulus generation and response checking, planning early on how to verify the design is crucial, as it affects their structure. The verification plan is directly derived from the specifications; it encompasses the description of what features shall be exercised and the techniques to do so.

Starting from it, tests are developed targeting those features, with the hope of uncovering discrepancies that designers may not have thought about. The typical steps are:

1. generate the stimulus
2. drive the DUT
3. capture the DUT's response
4. check the correctness of the response
5. track the progress in the overall verification plan

Depending on the chosen verification methodology, the way these steps are carried out changes.

1.2 Directed Testing

A typical straightforward approach is directed testing. Once having picked a certain feature to be verified from the plan, the testbench is constructed to generate enough stimulus vectors to exercise it satisfactorily. The DUT's response is collected in the form of a textual dump or as graphical waveforms and manually examined. If there are no violations the process advances and this is repeated until the verification plan is completed.

Although this approach starts yielding results quickly, each test must be written almost from scratch and the effort depends both on the complexity of the design and on its location in the system hierarchy. Furthermore, the stimulus vectors only exercise the DUT in expected areas, limiting the ability to uncover bugs elsewhere.

1.3 Find Bugs Faster

Considering the limited verification features available in VHDL and the simplicity of the designs tackled in introductory digital electronics courses, directed tests were sufficient.

In this first step of the *SoC Verification Strategies* workshop, I had the opportunity to start getting familiar with some common principles of more advanced methodologies and their elected language, SystemVerilog.

constrained-random stimuli Contrary to directed tests, which find bugs where they are expected to be, randomness allows to find bugs that were never anticipated; at the same time, constraints are essential to ensure that the stimulus is valid and relevant to the DUT.

functional coverage Once having switched to random tests, functional coverage becomes the metric for tracking progress in the verification plan, ensuring that all the intended features of the DUT were exercised.

layered structure Random stimuli imply the need for an environment capable of predicting the expected response; building this infrastructure requires additional work, thus the importance of effectively managing complexity:

- the abstraction level is raised up to the transaction level. The environment is structured in a layered manner, composing simpler modules.
- Language expressiveness limits analyzability, synthesizability and optimizability. Nonetheless, being verification the primary goal, Hardware Description Language (HDL)s make way for SystemVerilog and its convenient set of features, including:
  - constrained-random stimuli generation
  - functional coverage
  - high-level constructs borrowed from Object-Oriented Programming (OOP) and system programming
  - inter-thread communication and synchronization mechanisms
  - seamless integration with HDLs, especially with the event-based simulation kernel

Verification methodology libraries such as the Universal Verification Methodology (UVM) are devised for use by experienced users and excel on difficult problems. This first step of the workshop is dedicated to learning the background knowledge required to become proficient users of those more advanced libraries.

1.4 The Hands-On Experience

What follows is the analysis of the verification of two simple DUTs designed in the Microelectronic Systems course:

- as an example of a combinational circuit, the behavioral ALU presented in Section 2.2;
- as a sequential circuit, the behavioral accumulator presented in Section 2.3.

Both designs had been described in VHDL, which gave me the opportunity to experiment with mixed-language support of the commercial simulator [4].

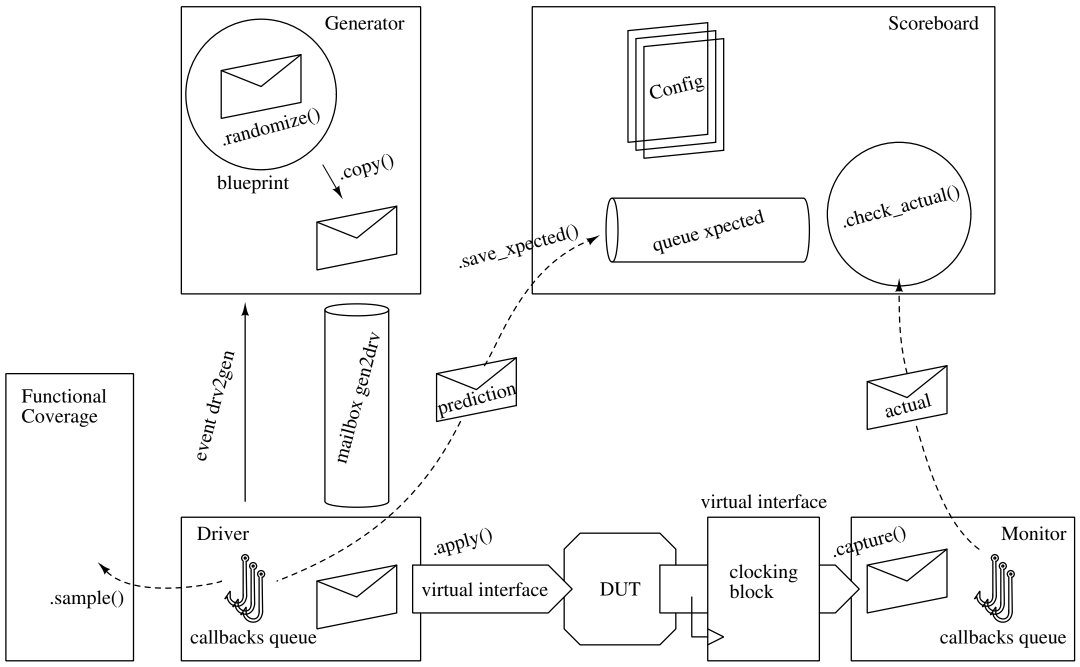


Figure 1: Conceptual representation of the layered testbench environment and its interface to the DUT. The dashed arrows highlight the callback calls; the letter envelopes mark the path followed by transactions around the testbench.

2 The Testbench Architecture

The architecture of the developed testbenches is conceptually represented in Fig. 1. The idea I had in mind sprouted from my previous experiences with C++ and a more recent exposure to SystemC. The goal was to exploit both OOP and generic programming capabilities of SystemVerilog to build a reusable, templated testbench framework that could later be specialized for specific DUTs. In the quest for already published works on the topic, I found [3] as a concrete example of what the final testbench environment could look like.

A significant novelty compared to general-purpose object-oriented languages is that data hiding is not stressed by the language; for instance, in C++ the default access to class members is private. I chose to stick with this default as a way to simplify code development.

2.1 The Core Components

The reusable core components are:

`virtual class BaseTransaction` Information gets passed around the testbench encapsulated in transaction objects. This abstract class is devised to be used as a base for inheritance and defines some fundamental methods to be used by other core components. For instance, `copy()` shown in Fig. 1, which gets called inside the `Generator` class to make copies of the blueprint transaction object.

```
class Generator
  #(type T = BaseTransaction);

  T blueprint;
  ...

  task run();
    T tr; // host the cloned blueprint

    repeat (n_tr) begin
      `SV_RAND_CHECK(blueprint.randomize()); // so to keep randomization history
      $cast(tr, blueprint.copy()); // then, copy

      tr.display($sformatf("@%0t: Generator: ", $time));

      gen2drv.put(tr); // send the transaction
      @drv2gen;        // the driver has finished with it
    end;

  endtask : run
endclass
```

Listing 2.1: The snippet highlights how type template parameters can improve code reusability, greatly reducing development time. `$cast()` here is an example of dynamic downcasting: `copy()` returns a base class handle that gets cast to point to the child class `T`.

`class Generator` The generator is responsible for creating random transactions and dispatching them to the driver through a mailbox, a communication channel that behaves like a FIFO.

The *blueprint* pattern is a convenient OOP technique here employed so that by changing this reference object it's possible to control the stream of generated transactions: the blueprint is repeatedly randomized, cloned and the clone gets sent through the mailbox. A significant reason for choosing to use this pattern is that, given that it's the blueprint object that gets randomized instead of the clone, the random-cyclic behavior of any `randc` member of the transaction class is respected.

The class is templated with the transaction type for flexibility, as shown in Listing 2.1.

`class Callback` Callbacks are a powerful mechanism in the testbench architecture, not only when it comes to reusing code for later projects, but also for developing a single verification environment that can be shared by all tests. To achieve that, there are "hooks" where tests or other environment modules can dynamically inject new code to execute.

`class Config` The config class serves as the central configuration descriptor for the testbench. In this case, for simplicity, I hardcoded the parsing of the command-line argument `+n_packets`, which allows changing the number of packets to generate to a value other than the default 10.

`class Scoreboard` The scoreboard acts as the central hub for collecting and verifying the expected and actual responses from the DUT. It connects to the driver and monitor modules via callbacks:

- the driver receives from the generator the stream of transactions to be applied to the DUT. After having applied each of them, it shall execute a hook to fill the transaction

```
package type_alu is
  type type_op is (add, sub, mult, bitand, bitor, bitxor, funclsl, funclsr, funcrl, funcrr);
end package;

entity alu is
  generic (n : integer);
  port (
    func:         in  type_op;
    data1, data2: in  std_logic_vector(n-1 downto 0);
    outalu:       out std_logic_vector(n-1 downto 0)
  );
end entity;
```

Listing 2.2: VHDL interface of the ALU under test. `type_op` is an enumerated type defined in a VHDL package, imported into SystemVerilog namespaces when necessary.

with the expected DUT's response and save it in the scoreboard.

- every time the monitor captures a response from the DUT, it shall execute a hook to make the scoreboard compare it with the expected one.

`virtual class BaseEnvironment` The environment is an abstract class that encapsulates all the blocks of the layered testbench and is devised as a base for inheritance. It simulates everything that is not inside the DUT, making it possible to run a certain testbench program via:

- `build()`, a method left to be implemented by child classes. It shall build the environment, which encompasses the allocation of the transactors and of the callbacks and their registration with both the scoreboard and the coverage class.
- `run()`. The generator, the driver and the monitor classes are run in their own threads. A timeout block prevents the simulation to hang in case of errors.
- `wrap_up()`, which prints a statistic of the current run, reporting the number of errors and the total functional coverage.

2.2 DUT - Arithmetic-Logic Unit

The ALU is designed to meet the following specifications:

- support for different word sizes
- support for various combinational operations:
  - *addition*, *subtraction* on full-width operands
  - *multiplication* on half-width operands
  - *and*, *or*, *xor* on full-width operands
  - *left/right shift* and *rotate* on a number of positions specified by the second operand.

```
constraint ab_dist_c {
    a dist {
      0                          := 10,
      [1:(64'd1<<DATA_WIDTH)-2] :/ 1,
      (64'd1<<(DATA_WIDTH-1))-1 := 10,
      (64'd1<<DATA_WIDTH)-1     := 10
      };
    b dist {
      0                          := 10,
      [1:(64'd1<<DATA_WIDTH)-2] :/ 1,
      (64'd1<<(DATA_WIDTH-1))-1 := 10,
      (64'd1<<DATA_WIDTH)-1     := 10
    };
  };
```

Listing 2.3: Weighted distribution constraint for the ALU input operands.

```
class AluDriver;
  v_alutb_if tb;               // Interface to the DUT
  ...

  // apply the stimulus
  task apply(input AluPacket pk);

    // synchronize
    @(tb.cb);

    tb.cb.op <= pk.op;
    tb.cb.a <= pk.a;
    tb.cb.b <= pk.b;

  endtask : apply

  ...
endclass

class AluMonitor;
  v_alutb_if tb;               // Interface to the DUT
  ...

  // capture the response
  task capture(output AluPacket pk);
    // allocate the packet where to store the response
    pk = new(.is_response(1));

    @(tb.cb); // synchronize
    pk.r = tb.r;

  endtask
  ...
endclass
```

Listing 2.4: Comparison between: driving the ALU inputs through the interface clocking block as synchronous signals. Reading the ALU outputs through the interface as an asynchronous signal but synchronized with the same active edge of the clocking block.

```
covergroup driver_packet_cg;
    op_cp : coverpoint pk.op; // automatically create bins for the enumerators

    a_cp : coverpoint pk.a {
      bins zero     = {0};
      bins max      = {(64'd1<<DATA_WIDTH)-1};
      bins others   = default; // ignored values for coverage
    }

    b_cp : coverpoint pk.b {
      bins zero     = {0};
      bins max      = {(64'd1<<DATA_WIDTH)-1};
      bins others   = default; // ignored values for coverage
    }
  endgroup
```

Listing 2.5: Functional coverage specifications for the ALU under test.

```
class AluScbDriverCb extends Callback#(AluPacket);
  ...
  virtual task post(input AluPacket tr);
    ...
    case (tr.op)
      /* arithemtic operations */
      add     : tr.r = tr.a  + tr.b;
      sub     : tr.r = tr.a  - tr.b;
      mult    : tr.r = tr.a[MultWidth-1:0] * tr.b[MultWidth-1:0];

      /* bitwise operations */
      bitand  : tr.r = tr.a & tr.b;
      bitor   : tr.r = tr.a | tr.b;
      bitxor  : tr.r = tr.a ^ tr.b;

      ...
    endcase

    scb.save_xpected(tr);
  endtask : post
endclass
```

Listing 2.6: Snippet of the golden model for the ALU under test, implemented as a driver callback.

2.2.1 The Testbench Refinement

The gap between the Register-Transfer Level (RTL) design and the SystemVerilog testbench is bridged by:

`package walu_pkg` This package holds the data parallelism configuration parameter and the related data type declaration.

`interface alu_if` As it can be observed in Fig. 1, the testbench wraps around the design, simulating everything that is not inside it.

In Verilog, the only possible approach for developing the interface would be to resort to ports, which would prove to be very error-prone when handling the hundreds of signals of complex designs. Furthermore, although separating design and testbench code each in its own module, oversights in the synchronization of the two can easily lead to race conditions around active edges of the clocks. The need for a higher-level way of communicating with the DUT is addressed by SystemVerilog's interface, a clever bundle of wires which simplifies:

- connectivity, since it can be instantiated like a module and connected to ports like a signal;
- synchronization, through *clocking blocks*. The clocking block is a non-synthesizable construct of the language that encapsulates the timing specifications of synchronous signals relative to the clocks.

`module walu` was defined to wrap the VHDL entity to make use of `alu_if`.

In addition to the core components, the testbench environment includes modules tailored to the ALU functionality, providing specialized transaction types, driver functionalities, response monitoring, and coverage analysis.

`class AluPacket extends BaseTransaction` As anticipated, the class refines the `BaseTransaction` in two main ways:

- given that an object of this class is used as blueprint by the generator, its data members shall undergo randomization. Once at the driver, the blueprint clones are unpacked and the data members values are used to drive the DUT. To make the randomization relevant, I chose to skew the inputs operands towards the corner cases, as visible in Listing 2.3.
- among the fundamental methods that the class shall implement there is `compare()`, used by the scoreboard to compare expected and actual DUT's responses.

`class AluDriver`, `class AluMonitor` An interesting aspect of the class that plays the role of applying the incoming transactions to the DUT's inputs is how the synchronization through the clocking block works. In this case, being the ALU combinational, there are no synchronous signals by design, but it's a policy that I found useful to synchronize operations within the testbench environment.

Similarly, for the class that captures the DUT's response, the clocking block comes in handy: it avoids having to pollute the code with repetitions of the clocking condition used for the synchronization. Look at Listing 2.4 for a comparison.

`class AluCoverage` The coverage class is dedicated to gathering information on how effectively the generated stimulus exercises the functionality of the ALU. In Listing 2.5 it can be seen how the coverpoints are defined to check for corner cases; in addition, being the operation input of an enumerated data type, it's terser to rely on the automatic bin generation capability of the language to check for operation type coverage.

`class AluScbDriverCb extends Callback` As anticipated when discussing callbacks, I chose this approach to equip the driver with the capability to generate the expected DUT's response, as a more readable alternative to SystemVerilog assertions, being it procedural code. A snippet is in Listing 2.6.

`class AluEnvironment extends BaseEnvironment` This class inherits from the `BaseEnvironment` abstract class, making it possible to instantiate it by implementing the `build()` method.

`program automatic alu_test` While discussing the `alu_if` I mentioned the possibility of accidentally generating race conditions when both the design and the test are encapsulated in modules. The root of the problem is that we would like design and testbench events to be not only logically but also temporally separated, which is not possible unless the even-based simulator has a clear understanding of the boundaries between the two. To address this problem, SystemVerilog changes the organization of the simulation time step, so that:

- the first region to execute in the time slot is the *active* one, where design events are run.
- It's followed by the *observed* region, where assertions are evaluated.
- "Then" there is the *reactive* region, where the testbench code located in program blocks is executed. Commonly to any event-based simulation, events in the observed and reactive regions can trigger further design events in the active region for the current time slot.
- Once all activity for the current time slot is extinguished, design signals are sampled in a read-only region termed *postponed*.

The `automatic` keyword serves the purpose of enabling automatic storage, whereas `static` is the default.

`module alu_top` This is the top module of the testbench, containing:

- the clock generator, which I chose to implement here being it more closely tied to the design domain of a digital system rather than to the testbench. In addition, this provides further separation between design and testbench events.
- The interface object.
- The DUT instance.
- The testbench program block.

2.3 DUT - Accumulator

The accumulator is designed to meet the following specifications:

- support for different word sizes
- selection of *sum* or *accumulate* operations via the *accumulate* signal, active high
- accumulation register with an active low enable signal, named *acc_ enable_ n*

```
entity acc is
  generic(
    numbit_g: positive
  );
  port (
    a:          in  std_logic_vector(numbit_g - 1 downto 0);
    b:          in  std_logic_vector(numbit_g - 1 downto 0);
    clk:        in  std_logic;
    rst_n:      in  std_logic;
    accumulate: in  std_logic;
    acc_en_n:   in  std_logic;
    y:          out std_logic_vector(numbit_g - 1 downto 0)
  );
end entity;
```

Listing 2.7: VHDL interface of the accumulator under test.

```
class AccScbDriverCb extends Callback#(AccPacket);
  ...
virtual task post(input AccPacket tr);

    if (!tr.rst_n)
      tr.y = 0;
    else if (tr.acc_en_n)
      tr.y = mem_y;
    else begin
      if (tr.acc) tr.y = tr.a + mem_y;
      else        tr.y = tr.a + tr.b;
    end

    mem_y = tr.y;          // update local memory element
    scb.save_xpected(tr); // save transaction
  endtask : post
endclass
```

Listing 2.8: Snippet of the golden model for the accumulator under test, implemented as a driver callback.

```
covergroup driver_packet_cg;

    // accumulator inputs,
    // when out of reset and not in memory state
    a_cp : coverpoint pk.a iff (pk.rst_n && !pk.acc_en_n) {
      bins zero     = {0};
      bins max      = {(64'd1<<DATA_WIDTH)-1};
      bins others   = default; // ignored values for coverage
    }

    b_cp : coverpoint pk.b iff (pk.rst_n && !pk.acc_en_n) {
      bins zero     = {0};
      bins max      = {(64'd1<<DATA_WIDTH)-1};
      bins others   = default; // ignored values for coverage
    }

    rst_n_cp : coverpoint pk.rst_n {
      bins reset_sequence = (1 => 0 => 1);
    }

    // accumulator states
    acc_cp : coverpoint pk.acc iff (pk.rst_n) {
      option.weight = 0; // don't count this coverpoint alone
    }
    acc_en_n_cp : coverpoint pk.acc_en_n iff (pk.rst_n) {
      option.weight = 0; // don't count this coverpoint alone
    }

    cross acc_cp, acc_en_n_cp {
      bins memory_state     = binsof(acc_en_n_cp) intersect {1};

      bins sum_state        = binsof(acc_cp) intersect {0} &&
                              binsof(acc_en_n_cp) intersect {0};

      bins accumulate_state = binsof(acc_cp) intersect {1} &&
                              binsof(acc_en_n_cp) intersect {0};
    }
  endgroup
```

Listing 2.9: Functional coverage specifications for the accumulator under test. Being it a sequential circuit, there are some additional cases worth reporting to measure how extensively the functionality of the DUT was tested. In the cross-coverage specification, the `acc_cp` and `acc_en_n_cp` cover-points would automatically originate 4 bins: here the two bins corresponding to `acc_en_n_cp == 1'b1` are merged because both are associated to the case of the output register of the accumulator being disabled.

2.3.1 The Testbench Refinement

As clarified when introducing the advantages of having a higher-level testbench framework, changing the DUT no more implies having to rewrite the testbench from scratch. There's a one-to-one mapping with the components already developed for the ALU, thus I'm highlighting only the novelties here.

`class AccCoverage` With a sequential circuit there are a few additional cases that make the stimulus interesting and thus worth reporting:

- the reset sequence,
- the possible states for the accumulator: *memory*, *sum* and *accumulate*.

The former is checked by means of the transition coverage syntax; the latter by means of cross-coverage, using custom-defined bins. The relevant snippet is in Listing 2.9

`class AccScbDriverCb extends Callback` Also in this case I chose not to use SystemVerilog assertions; a callback takes care of implementing the golden model with procedural code and a class data member plays the role of the storage element, as it's visible in Listing 2.8.

`program automatic acc_test, module acc_top` Differently from the simpler combinational ALU, here it's essential that the DUT has been successfully reset before running the environment. The synchronization issue is solved with a naked event, shared with the top module.

3 Results

The results presented in the following are based on simulations performed using the commercial simulator [4]. The output logs were then post-processed with custom *tcl* scripts, run within the simulator shell, and with MATLAB scripts.

3.1 Arithmetic-Logic Unit

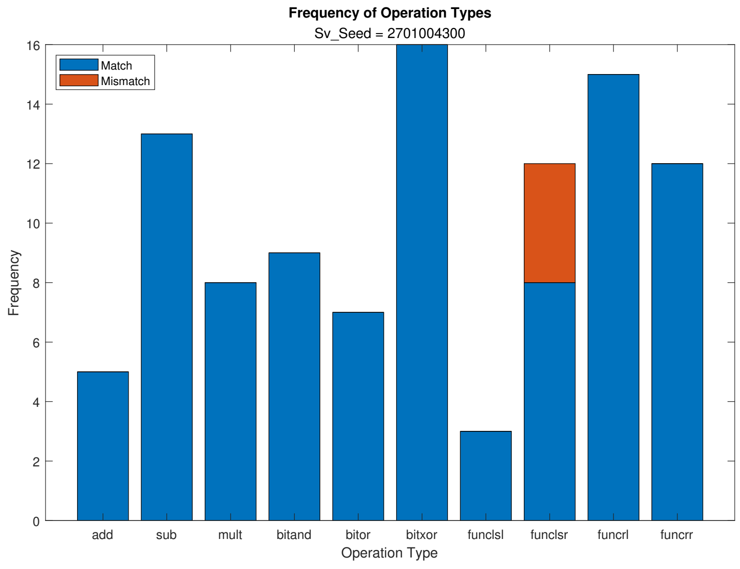
The ALU was initially tested using a random seed, data parallelism of 32 bit and 100 packets. The simulation log prints:

```
# @1010: Scoreboard: 100 expected packets, 100 received packets
#
# @1010: END OF SIMULATION
#          * 4 error(s)
#          * total functional coverage: 100.00%
```

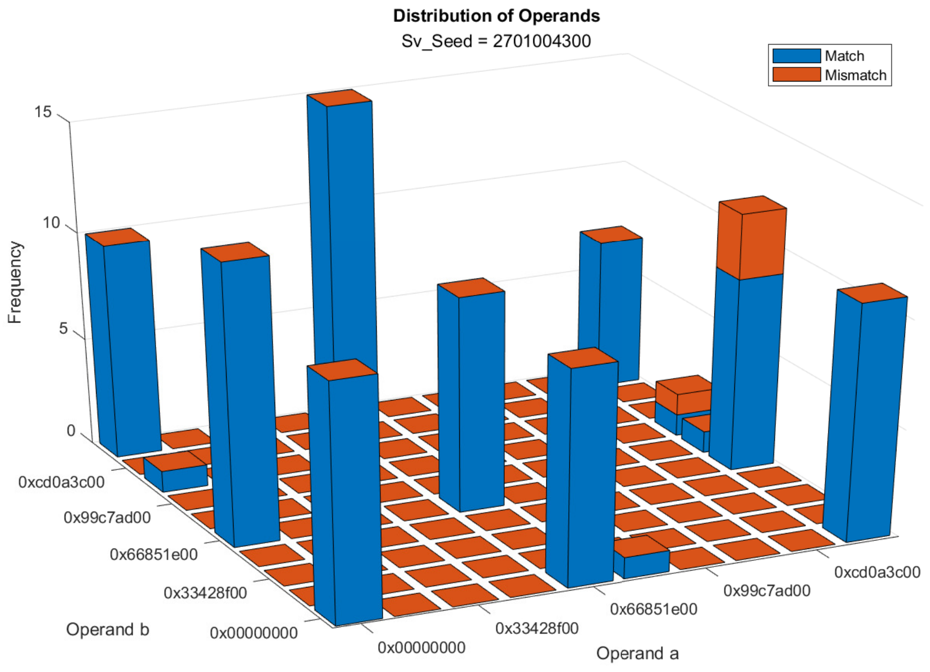
The representation of the output is in Fig. 2.

Impersonating both the design team and the verification team, the additional knowledge about the RTL description gives an advantage in the understanding of what could have gone wrong. Before that, I can highlight a first shortcoming in the definition of the cover group: it would be very useful to have an insight into the distribution of the operand corner cases per operation type. Otherwise, I would have to compute them in post-processing.

Doubting that the stimulus was sufficient to exercise all operation types in the corner cases, I repeated the simulation increasing the packets number to 1000; the results in Fig. 3 prove that both *funclsr* and *funclsl* are faulty.



(a) The histogram shows that the applied stimulus was effective for exercising all the operations supported by the DUT. The *funclsr* is faulty, having generated wrong results in 1/3 of the cases.



(b) This bar plot allows us to appreciate how the realizations of the input operands are distributed in the support set. In compliance with the chosen weighted distribution, the stimulus is skewed towards the corner cases: this is where the 4 errors of the *funclsr* operation are located.

Figure 2: Visualization of the simulation logs for the ALU under test: 32 bit parallelism, 100 packets.

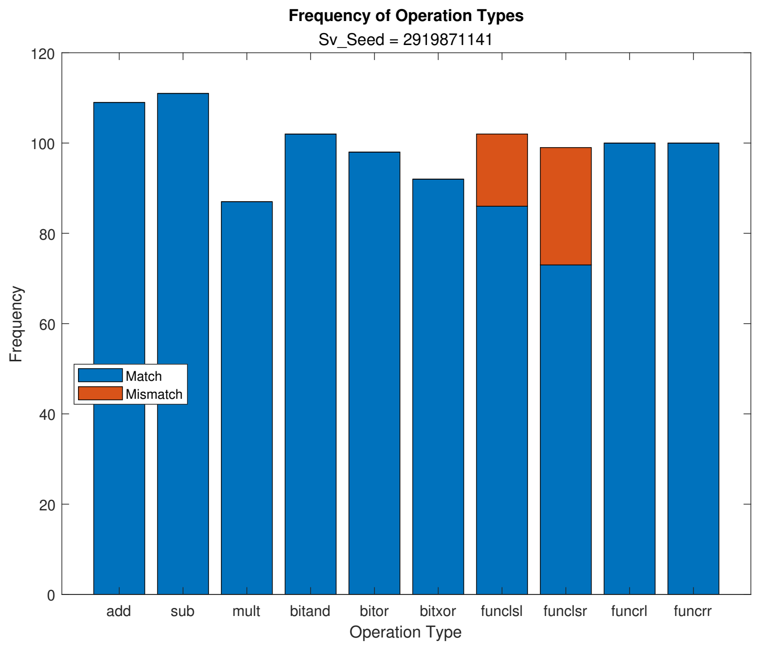


Figure 3: Additional run of the ALU simulation: 32 bit parallelism, 1000 packets. Both *funclsr* and *funclsl* are faulty.

3.1.1 Debugging

Looking at the offending lines in the behavioral description of the ALU:

```
when funclsl =>
    outalu <= std_logic_vector(shift_left(unsigned(data1), to_integer(unsigned(data2))));
when funclsr =>
    outalu <= std_logic_vector(shift_right(unsigned(data1), to_integer(unsigned(data2))));
```

I observe that the second operand is not sliced to restrict the maximum number of shift locations inside `[0:data_width-1]`. Now, looking at the details in the simulation log for one of such errors:

```
# ** Warning: (vsim-151) NUMERIC_STD.TO_INTEGER: Value -2 is not in bounds of subtype NATURAL.
#    Time: 10 ns  Iteration: 2  Instance: /alu_top/dut/alu_i
# @20: Monitor: Packet id=2: { r=fffffffe }
# @20: Scoreboard: check: Packet id=2: { r=fffffffe }
#                 against: Packet id=1: { a=7fffffff, b=fffffff, r=00000000, op=funclsr }
#                 ERROR MISMATCH
```

Searching the `ieee` library[1], the function that gets called is:

[1]`Revision: #1, Date: 2020/10/03`

```vhdl
-- Id: D.1
function TO_INTEGER (ARG : UNRESOLVED_UNSIGNED) return NATURAL is
  constant ARG_LEFT : INTEGER := ARG'length-1;
  alias XXARG       : UNRESOLVED_UNSIGNED(ARG_LEFT downto 0) is ARG;
  variable XARG     : UNRESOLVED_UNSIGNED(ARG_LEFT downto 0);
  variable RESULT   : NATURAL := 0;
begin
  if (ARG'length < 1) then
    assert NO_WARNING
      report "NUMERIC_STD.TO_INTEGER: null detected, returning 0"
      severity warning;
    return 0;
  end if;
  XARG := TO_01(XXARG, 'X');
  if (XARG(XARG'left) = 'X') then
    assert NO_WARNING
      report "NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0"
      severity warning;
    return 0;
  end if;
  for I in XARG'range loop
    RESULT := RESULT+RESULT;
    if XARG(I) = '1' then
      RESULT := RESULT + 1;
    end if;
  end loop;
  return RESULT;
end function TO_INTEGER;
```

The problem originates from the conversion loop: the underlying data type is not wide enough, therefore `to_integer(X"ffffffff")` conversion fails to return 4 294 967 295. In fact, the package `standard`[2] defines:

```vhdl
type INTEGER is range -2147483648 to 2147483647;
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
```

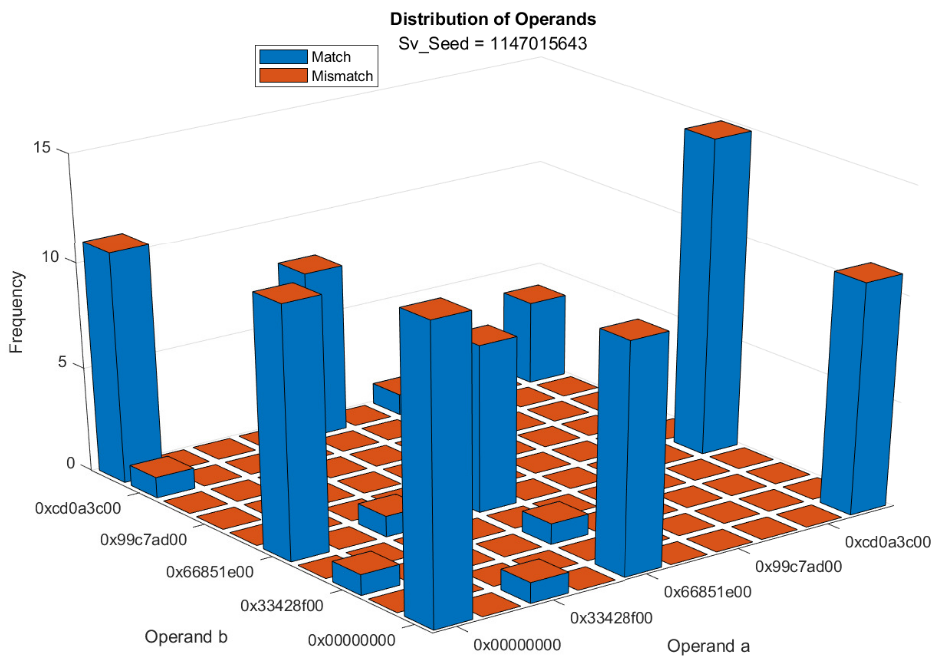
3.2 Accumulator

Similarly, the accumulator was initially tested using a random seed, data parallelism of 32 bit and 100 packets. The simulation log prints:

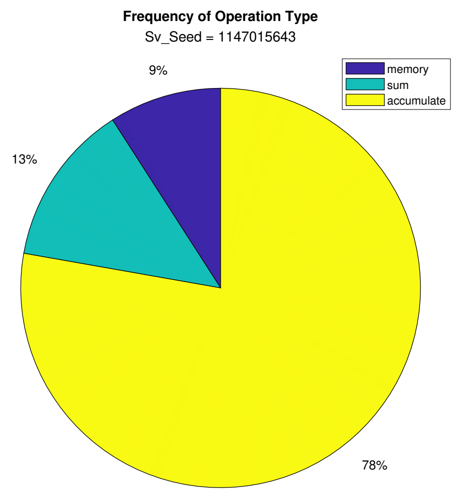
```
# @1020: Scoreboard: 100 expected packets, 100 received packets
#
# @1020: END OF SIMULATION
#           * 0 error(s)
#           * total functional coverage: 100.00%
```

The representation of the output is in Fig. 4. Specifically, the cross-coverage data is represented in the pie chart of Fig. 4b and allows us to appreciate how long the DUT operated in its different states. On the other hand, Fig. 4a highlights how the weighted distribution constraint effectively skewed the stimulus towards the boundaries of the support set.

[2]VHDL 2008 Language Reference Manual



(a) The bar plot highlights that no violations occurred; as in the case of the ALU, input operands are localized in the corner cases.



(b) In the pie chart we can observe the relative time the DUT spent in each one of the possible states.

Figure 4: Visualization of the simulation logs for the accumulator under test; 32 bit parallelism, 100 packets.

4 Conclusion

In this analysis, I explored the importance of early bug detection in the design cycle and the crucial role verification engineers play in ensuring compliance with design specifications. The shortcomings of the directed testing approach are overcome with more advanced methodologies, built upon the features of the SystemVerilog language. Constrained-random stimuli, functional coverage and code reuse through high-level abstractions are essential tools for managing complexity and achieving results throughout the verification plan.

Standardized verification methodologies target an experienced user base, already proficient in the way various programming paradigms can help create a highly reusable and flexible testbench framework and aware of the many limitations of directed tests. This work can be seen as an introductory step, providing a bridge between traditional directed testing approaches and the adoption of more advanced methodologies, by leveraging simple object-oriented and generic programming principles. Furthermore, the discussed testbench architecture is general enough to be easily adaptable to typical designs of digital systems courses.

As an example of effectiveness, the hands-on experience with the ALU unearthed a bug that went overlooked in simpler directed tests, which was partly tied to the representation width of native VHDL integer types.

References

[1] "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language," *IEEE Std 1800-2005*, pp. 1–648, 2005. DOI: 10.1109/IEEESTD.2005.97972.

[2] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006. DOI: 10.1109/IEEESTD.2006.99495.

[3] C. Spear and G. Tumbush, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd. Springer US, 2012, ISBN: 9781461407140 (cit. on p. 5).

[4] Mentor Graphics Corporation, *Questasim-64*, version 2020.4 linux_x86_64, Oct. 13, 2020 (cit. on pp. 4, 14).

Acronyms

ALU	Arithmetic-Logic Unit
DUT	Device Under Test
HDL	Hardware Description Language
OOP	Object-Oriented Programming
RTL	Register-Transfer Level