Politecnico di Torino

Department of Electronics and Telecommunications

Multidisciplinary Workshop *SoC Verification Strategies*

Step 2: Technical Report

Fabio Scatozza: s315216@studenti.polito.it

September 1, 2023

Contents

List of Listings

List of Figures

1 Introduction

In the *Introduction to SystemVerilog* step, I had the opportunity to tackle the limitations of the directed testing approach, getting familiar with some common principles of more advanced verification methodologies and their elected language, SystemVerilog. Primarily guided by [4], I applied object-oriented and generic programming principles to craft a reusable, templated testbench framework that I could specialize for specific DUTs. Through the design and development of the entire testbench architecture, I gained a better understanding of the techniques that go into the classes and utilities of the Universal Verification Methodology (UVM) library, which established itself as a de facto industry standard for functional verification.

1.1 The Universal Verification Methodology

The complexity of digital designs continues to escalate, posing tough challenges to ensure their correctness and reliability. Curated by the Accelera UVM Working Group, this methodology focuses on establishing modularity, scalability, and reusability in verification environments, with the core objective of augmenting design productivity. Both the standard [6] and the class library are available free of charge; the library is conveniently shipped in the package `uvm_pkg`.

1.1.1 Base Classes: An Overview

The basic building blocks for all environments are components and the transactions they use to communicate.

- All components and transactions derive from `uvm_object`, an abstract base class whose primary role is to define an interface for common operations as `create`, `copy`, and `compare`. It also defines interfaces for instance identification, which includes name, type name, id, and seeding.
- Components are objects derived from `uvm_component` that exist throughout the simulation. They establish a structural hierarchy, like modules and program blocks, and take part in the execution under the control of a central phasing mechanism; for each phase, they define a callback function or task that is executed in a precise order.
- Unlike components, transactions are transient in nature. In the simpler cases, they can be derived directly from `uvm_transaction`, but for operating in the sequence mechanism, the base class is `uvm_sequence_item`.

1.1.2 The UVM Architecture

The typical architecture of a UVM-based testbench is shown in Fig. 1. For each interface of the Device Under Test (DUT) there is a dedicated verification component called agent, which includes:

- The driver, whose job is to communicate at the signal level with the DUT through the interface, translating incoming sequence items to pin wiggles.
- The sequencer, which arbitrates the flow of sequence items to the driver, possibly routing its responses back to the correct generator sequence. On a given sequencer it's possible to have multiple sequences executing, which is why it supports arbitration. Sequences are objects whose job is to generate a stream of sequence items.

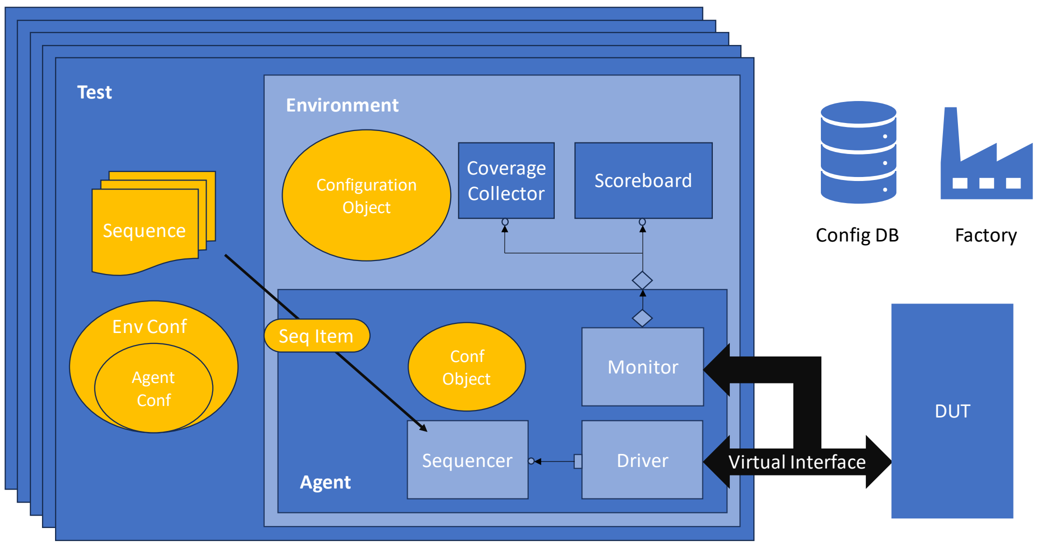


Figure 1: Typical UVM-based testbench architecture. Each interface of the DUT is served by an agent: both the driver and the monitor communicate at the signal level, whereas any other communication is at the transaction level. A library of test components can use the same environment, possibly customizing it via factory overrides and the configuration database.

- The monitor, which recognizes the pin-level activity on the interface and turns it into transactions that get broadcasted to other environment components through an analysis port.
- A configuration object, which contains all the information the agent needs to know how to operate; for instance, the virtual interface handle.

The environment acts as a container of interrelated verification components; it gets its configuration object first and uses it to configure itself and the components it instantiates. Typically, these additional blocks include a coverage collector, which tracks progress in the verification plan, and the scoreboard, which checks DUT responses to the randomized stimulus. In turn, the environment is instantiated by a test component: it specifies the sequences to execute and the customization of the environment both in terms of factory overrides and configuration objects stored in the database. It is common to develop a base test class and craft a library of extended tests that only add sequences and stimulus-specific customizations.

1.2 The Hands-On Experience

What follows is the analysis of the UVM-based verification of two intermediate designs from the *Microelectronic Systems* course.

- As an example of a combinational circuit, Intel's Pentium IV adder. According to the interface in Listing 1.1, it is designed both with generic parallelism and carry-generator tree sparseness.

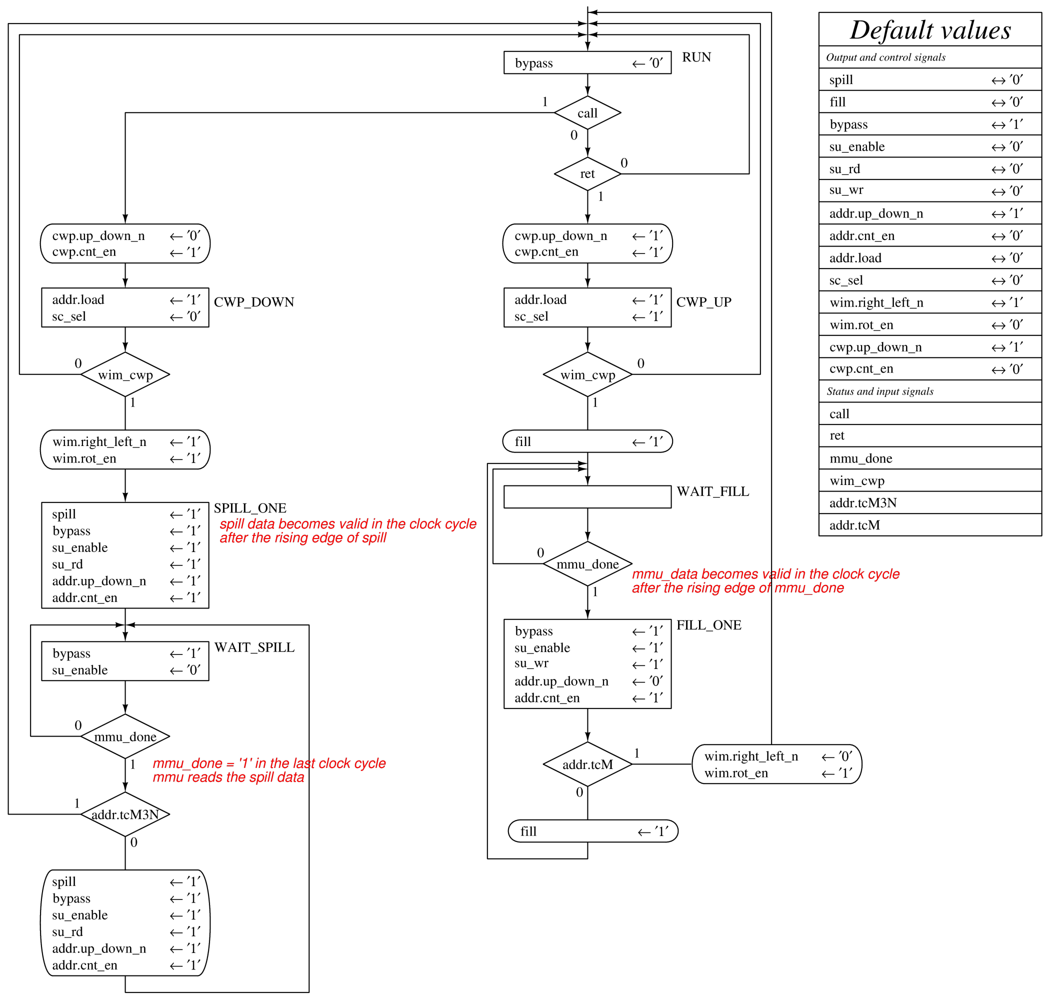
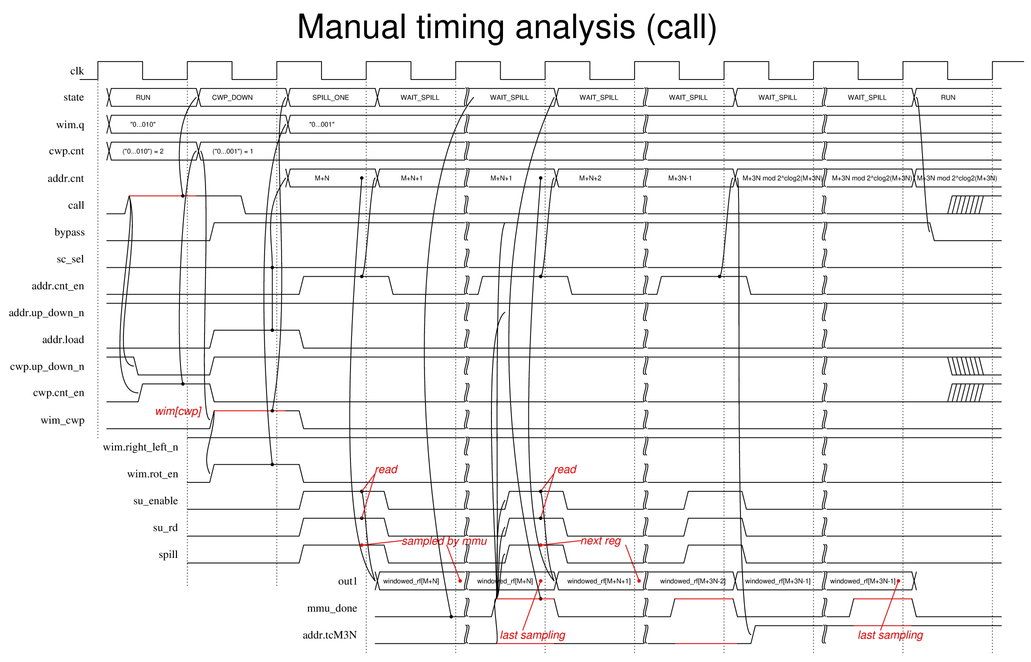


Figure 2: Windowed register file ASM chart, describing the signal-level details of managing call and return requests. When both the `call` and `ret` signals are asserted, the call request takes precedence. In both cases, the `bypass` signal is raised while the register file is busy. If the request triggers a spill or fill operation, the `spill` signal is raised with a one-cycle delay after the `bypass` signal, whereas the `fill` signal is raised within the same cycle. The memory management unit must set the `mmu_done` signal according to specifications: in the last clock cycle it reads data on `out1` for a spill operation; in the clock cycle before the data on `mmu_data` becomes valid for a fill operation.



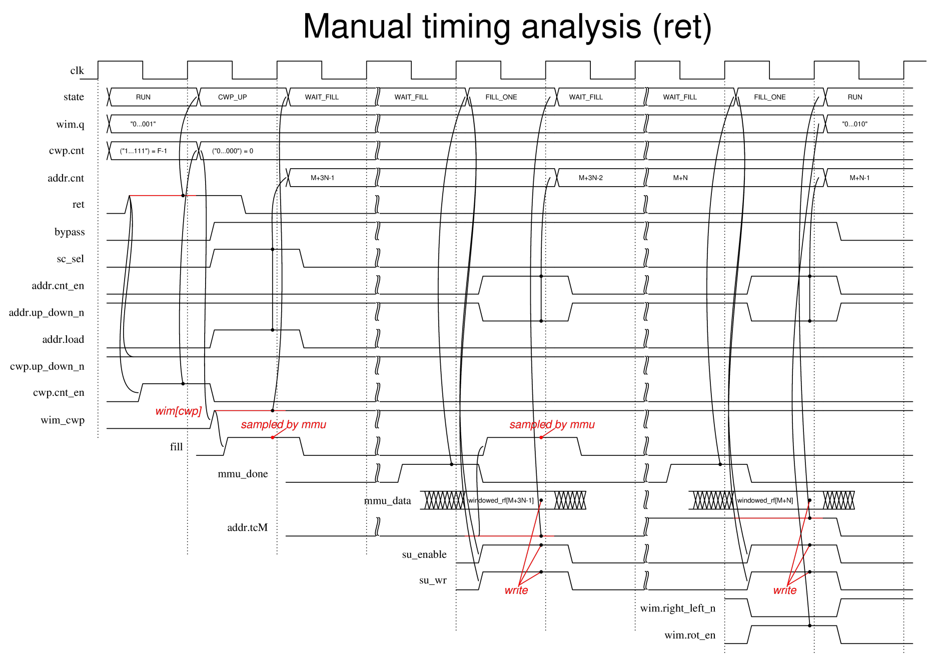


Figure 3: Timing details of call and return requests.

```
entity p4_adder is
  generic (
    nbit:          integer := 32;
    nbit_per_block: integer := 4);
  port (
    a:    in    std_logic_vector(nbit-1 downto 0);
    b:    in    std_logic_vector(nbit-1 downto 0);
    cin:  in    std_logic;
    s:    out   std_logic_vector(nbit-1 downto 0);
    cout: out   std_logic
  );
end entity;
```

Listing 1.1: VHDL interface of Intel's Pentium IV adder under test.

- As a sequential circuit, the fixed-size windowed register file of Listing 1.2, inspired by the SPARC instruction set architecture [1]. The added difficulty of the overlapping windows organization is the management of call and return requests, whose control was designed according to the Algorithmic State Machine (ASM) methodology as shown in Fig. 2. For the interaction with the memory management unit, a simple protocol was chosen; the detailed timing is in Fig. 3.
  - In the clock cycle after having sampled `call` high, the `bypass` signal is raised; if the call request triggers a spill operation, the `spill` output is raised in the subsequent cycle and the spilled data becomes valid on `out1` from the cycle onward. The memory management is expected to respond by raising the `mmu_done` signal in the last clock cycle of data reading.
  - In the clock cycle after having sampled `ret` high, the `bypass` signal is raised; if the return triggers a fill operation, the `fill` signal is also raised within the same cycle. The memory management unit recognizes the fill request by monitoring the `fill` signal and responds by raising `mmu_done` in the clock cycle before the data becomes valid on the `mmu_data` input to the register file.

  The reset is synchronous and prevails on call and return requests; in turn, the call request holds higher priority over the return. In the absence of these three, the `enable` signal activates read and write operations, which must be further enabled port by port through the `rd1`, `rd2`, and `wr` signals. In cases where the read and write addresses match, a read-before-write policy is adopted.

2 The Verification Plan

Starting from the design specifications I formulated the verification plans, which outline the test cases for the features to be exercised; the techniques chosen to measure progress in the verification process are both code and functional coverage.

Given that both the DUTs are parameterized, a simple development strategy involves maintaining a package that defines the DUT namespace, within which declaring the global parameters, whose value is set via macros defined at compile time through the command line. Once having identified the range of interest for the parameters, this approach only requires repeating the simulations while changing the macros definitions.

```vhdl
entity windowed_rf is
  generic (
    M:          positive := 8;  -- number of global registers
    N:          positive := 8;  -- number of registers per register subset
    F:          positive := 8;  -- number of register sets (N "in" + N "local")
    width_data: positive := 64  -- parallelism of data
  );
  port (
    clk:      in  std_logic;
    reset:    in  std_logic;  -- synchronous, write-through
    enable:   in  std_logic;  -- gates rd1, rd2, wr

    -- 2R / 1W ports
    rd1:      in  std_logic;  -- enables synchronous reading on port R1
    rd2:      in  std_logic;  -- enables synchronous reading on port R2
    wr:       in  std_logic;  -- enables synchronous writing

    -- Addressing convention:
    -- Windowed Register Address  Register Address
    --   in[0] - in[N-1]            r[M+2*N] - r[M+3*N-1]
    --   local[0] - local[N-1]      r[M+N] - r[M+2*N-1]
    --   out[0] - out[N-1]          r[M] - r[M+N-1]
    --   global[0] - global[M-1]    r[0] - r[M-1]

    add_rd1:  in  std_logic_vector(clog2(M+3*N)-1 downto 0);
    add_rd2:  in  std_logic_vector(clog2(M+3*N)-1 downto 0);
    add_wr:   in  std_logic_vector(clog2(M+3*N)-1 downto 0);

    datain:   in  std_logic_vector(width_data-1 downto 0);
    out1:     out std_logic_vector(width_data-1 downto 0);
    out2:     out std_logic_vector(width_data-1 downto 0);

    -- execution flow control
    call:     in  std_logic;  -- subroutine call
    ret:      in  std_logic;  -- subroutine return
    bypass:   out std_logic;  -- '0' when available for external operations

    -- interface to memory management unit
    fill:     out std_logic;
    spill:    out std_logic;

    mmu_done: in  std_logic;  -- fill/spill synchronization
    mmu_data: in  std_logic_vector(width_data-1 downto 0)
  );
end entity;
```

Listing 1.2: VHDL interface of the fixed-size windowed register file under test.

```
`define ASSIGN_UNKNOWN_CHECK(lhs, rhs) \
  do begin \
    lhs = rhs; \
    if ($isunknown(rhs)) \
      uvm_report_warning("capture", "dut outputs unknown bits"); \
  end while (0)
```

Listing 2.1: Macro function to guard against the propagation of unknown values when reading the DUT outputs.

To improve simulation performance and reduce the memory footprint, I opted for the 2-state data types offered by SystemVerilog. This requires additional care when connecting the testbench to the outputs of the DUT, as it may try to drive `x` or `z` values that would be automatically converted to a 2-state value. The check is performed with the macro function shown in Listing 2.1, defined in both DUTs namespaces.

2.1 Functional Coverage Collection

As shown in the testbench architectures of Fig. 4, the response transactions broadcasted by the monitor through the analysis port reach the coverage collector subscriber, a component that extends `uvm_subscriber`, parameterized by the response transaction class `RspTxn`, that provides an implementation for the port function `write()`. The coverage collector is implemented in a hierarchy of two levels:

`virtual class Coverage` It's the abstract base class handled by the environment, thus meant to be overridden by test classes via the factory. It implements the `write()` function of the monitor analysis port: it grabs the incoming response transaction in a local handle, then it invokes the pure virtual function `sample()`.

`class StmCoverage` It's the child class included in each DUT namespace that embeds the cover group for the test cases of interest and implements the `sample()` function to let the parent trigger the sampling of the cover group.

2.2 Intel's Pentium IV Adder

I selected the following test cases as the basis for functional coverage:

1. Zero input:
    - 1.1. All 0s on an input
    - 1.2. All 0s on both inputs
2. One input:
    - 2.1. All 1s on an input
    - 2.2. All 1s on both inputs
3. Non-corner values on both inputs
4. 0s on both inputs and carry-in 0

```
a_cp : coverpoint txn.a {

  bins zeros  = { 0 };
  bins others = { [1:{NBIT{1'b1}}-1] };
  bins ones   = { {NBIT{1'b1}} };

  /* don't count the coverpoint alone */
  type_option.weight = 0;
}

b_cp : coverpoint txn.b {

  bins zeros  = { 0 };
  bins others = { [1:{NBIT{1'b1}}-1] };
  bins ones   = { {NBIT{1'b1}} };

  /* don't count the coverpoint alone */
  type_option.weight = 0;
}

a_cp_cross_b_cp : cross a_cp, b_cp {

  /* testcase 1.1 */
  bins one_zeros   = (binsof(a_cp.zeros) &&
                       (binsof(b_cp.others) || binsof(b_cp.ones))) || // a zeros, b not
                     (binsof(b_cp.zeros) &&
                       (binsof(a_cp.others) || binsof(a_cp.ones)));   // b zeros, a not
    ...
}
```

Listing 2.2: Snippet of the cross coverage statements used to measure coverage for the test cases in the verification plan of the adder under test.

5. 1s on both inputs and carry-in 1
6. (unsigned) overflow

The translation to a SystemVerilog cover group is done in two steps. First, I defined cover points to observe the values of the data inputs `a` and `b` with custom bins that isolate the corner cases *all 0s* and *all 1s*. In particular, I used the option `type_option.weight` to avoid counting these auxiliary cover points and the `cin` cover point alone. Then, I used cross coverage to combine the cover point bins according to the test cases of interest; the Listing 2.2 shows test case 1.1 as an example.

2.3 Fixed-Size Windowed Register File

I selected the following test cases as the basis for functional coverage:

1. Execute all operations. In the case of read and write operations, execution means that the operation must be issued and must not be masked by a reset, a call, or a return.
2. Read and write operations:
   2.1. Disabled by `rd1`, `rd2` or `wr` while enabled by `enable`

```
typedef struct packed {
  bit rd1; // msb
  bit rd2;
  bit wr;
  bit call;
  bit ret;
  bit enable;
  bit reset;
} packed_ops_t;

/* testcase 1
 * notice: call, return and reset can mask read and write operations */
execute_rw_cp : coverpoint txn.get_ops() {
  wildcard bins rd1   = { 7'b1??0010 };
  wildcard bins rd2   = { 7'b?1?0010 };
  wildcard bins wr    = { 7'b??10010 };
}
/* testcase 1 */
execute_call_ret_reset_cp : coverpoint txn.get_ops() {
  wildcard bins reset       = { 7'b??????1 };
  wildcard bins call        = { 7'b???1??0 };       // reset wins over call
  wildcard bins ret         = { 7'b???01?0 };       // reset and call win over ret
}
```

Listing 2.3: The snippet shows a more readable and less error-prone approach to cross coverage when the number of cover points is large; it's mimicked by packing the variables as bit fields in a vector and selecting the combinations of interest using wildcard values. Notice that `get_ops()` is a method of the request transaction class that returns a `packed_ops_t`.

   2.2. Enabled by `rd1`, `rd2` or `wr` but disabled by `enable`
   2.3. Issued but masked by a reset, a call, or a return
   2.4. Read-before-write, distinguishing between read ports 1 and 2

3. Call and return operations:
   3.1. Issued together
   3.2. Issued with a reset
   3.3. Executed twice in a row
   3.4. Generate a spill
   3.5. Generate a fill

4. Reset operation:
   4.1. Executed twice in a row
   4.2. Execute all operations, except for reset and return, after a reset
   4.3. Execute all operations, except for reset, before a reset

Unlike the simpler adder, the translation to a SystemVerilog cover group is not primarily done with cross coverage, but with an approach that mimics it. The request transaction class `RqstTxn` provides the method `get_ops()` to export the bits that define the requested operations in the packed struct `packed_ops_t`, which is treated as if its members were concatenated in a single vector. A cover point records the observed values of a single variable or expression, which means that concatenations and packed structs are allowed as well: to make the definition of the combinations terser, SystemVerilog provides the `wildcard` keyword to specify the bins with bit patterns including don't care values. The Listing 2.3 shows test case 1 as an example.

Test cases 3.3 and 4.1 are handled with the consecutive repetition operator, whereas test cases 4.2 and 4.3 are covered by combining wildcard states and the transition operator.

3 The Testbench Architecture

Compared to the development process of the previous introductory step, the adoption of the UVM allowed me to move past the low-level intricacies of setting up the testbench environment with all the components and utilities I could need, and focus instead on a smaller set of DUT-specific problems; most notably:

- Interfacing the DUT and the environment through the agent
- Crafting the reference model to validate the DUT responses to the random stimulus
- Devising the functional coverage model to measure progress
- Configuring the reporting system to easily post-process the results of the simulation runs

This resulted in faster and higher quality development, as I was taking advantage of an already extensively tested code base and I could direct the saved time and efforts toward the most challenging problems of the designs I had to verify. The conceptual representation of the testbenches is shown in Fig. 4, with the detailed interface for the windowed register file in Fig. 5

Each testbench project can be partitioned into three core components:

1. A SystemVerilog interface to bundle the DUT wires. It simplifies both connectivity, as it can be instantiated like a module and connected to ports like a signal, and synchronization, through clocking blocks that encapsulate the timing specifications of signals that are read or driven by the testbench synchronously to a clock.

2. A SystemVerilog package to collect the verification classes and utilities, similar to the `uvm_pkg`. This simplifies source management, given that all the definitions and declarations pertaining to a testbench are available in a package ready to be imported into the top module.

3. A top-level module and program block. The program block ensures race-free interaction between the design and the testbench at the simulation kernel level and has the fundamental job of starting the execution of the UVM test; it is instantiated taking as input the interface, which is shared through virtual handles in the configuration database to the monitors and drivers.

   The top module contains the instances of the DUT, the interface, and the program block. In addition, it hosts the clock generator, that I chose to implement here, being it more closely tied to the design domain rather than to the testbench.

3.1 Reusability

A key idea to reusability is to separate structure from behavior. Being transactions, or specifically sequence items, the main communication vehicle across the boundary, their modeling has paramount importance and should come early in the development flow. Moreover, to maximize code reuse and further decrease the bug rate, I strived to take advantage of inheritance to avoid code duplication between the two projects.

3.1.1 Sequence Item Design

In both DUT namespaces I defined:

`class RqstTxn` Having chosen to adopt the sequence mechanism, the request transaction is defined to extend `uvm_sequence_item` and has two main purposes:

- It's created through the factory by sequences and randomized to generate the stream of sequence items to feed the driver. Its data members are the ones the driver translates to pin wiggles onto the signals in its modport and are qualified to undergo randomization.

  I avoided adding randomization constraints in this class, with the idea that tests can use the factory and override this type with a child class that includes test and stimulus-specific constraints; specifically, `CnstRqstTxn`. The advantage is that in case simulation runs had highlighted coverage holes, I would've been able to add tests with different constraints to target the remaining cases.
- It serves as the parent for the response transaction class. Given that the scoreboard needs to compare the expected and actual DUT responses, to add robustness I implemented the method `do_compare()`, called internally by `compare()`, to preliminary assess whether the responses pertain to the same request or not.

Notice that to improve simulation performance I always print transaction objects through the UVM report functions passing the string generated by `convert2string()`. The alternative would be to implement `do_print()` and invoke `print()`, but it introduces a significant overhead.

`class RspTxn` It's the transaction used in analysis communication, broadcasted by the monitor, outside the agent and to subscriber components in the environment. It encapsulates the data members worth capturing with the monitor: the request fields are inherited by extending `RqstTxn` and having both the request and response fields available in the broadcasted transactions is crucial for scoreboard operation. This is because I chose to embed the reference model that processes requests and generates expected responses within the scoreboard.

This approach has a downside that became evident when tackling the windowed register file: when request-response pairs are collected in an overlapping fashion, if it's possible for the current request to abort the previously sampled one, whose response is currently being sampled, a mechanism to inform the scoreboard is needed. As a consequence, for the windowed register file, `RqstTxn` is extended by `RqstAnlysTxn`, which adds a field to mark the request as aborted, and the latter is then extended by `RspTxn`.

3.1.2 Shared Base Classes

Although agents share a common structure for stimulus generation and response capturing, the UVM library already ships the necessary abstractions for its implementation and the number of base classes to be manually developed and shared between projects is minimal.

First, it's good practice to avoid directly using the `uvm_sequencer` class and define an extended class registered with the factory that will be displayed when printing the testbench topology; this is done with `class Sequencer`. For the components outside of the agent, in addition to the base coverage collector class described in Section 2.1, there are:

`class Printer` The printer listens on the analysis port and prints the incoming transactions both to the screen and to a file log. The key aspects of file logging are that the printer retrieves the file path from the configuration database in the build phase; then, in the end-of-elaboration phase, it associates the id `"printer"` to the printer file descriptor and to the action `UVM_DISPLAY | UVM_LOG`. On the contrary, `class BitBucket` extends `Printer` and sets as action `UVM_LOG`, to only print transactions to the log file.

If debugging messages are enabled, by raising the verbosity at least to `UVM_HIGH`, the printer reports a state dump in case of a successful configuration.

`virtual class BaseScoreboard` It's the abstract base class that implements those anticipated common duties of scoreboards as response checkers, but are the child classes the ones that implement the pure virtual function `generate_prediction()` based on a reference model. Like a printer, it listens on the analysis port to incoming transactions:

- in the `write()` callback function, it clones the incoming DUT response for its request fields and generates the expected response. If the comparison yields a mismatch, an error is reported both on screen and to the scoreboard log file; otherwise, the match is only logged to file.

  The only difference in the file logging management from the printer is that, in the end-of-elaboration phase, the id `"scoreboard"` is associated with different actions depending on the severity, so that only mismatch reports with severity `UVM_ERROR` reach the screen.

- if the job of stopping the simulation was entirely left to sequences, which drop objections as soon as the driver notifies `item_done()` on the last sequence item, the DUT wouldn't be able to respond in time and, consequently, also the monitor and its subscribers. The solution I implemented is the following. In the build phase, `BaseScoreboard` clears a flag that only gets set once the number of expected responses matches the ones actually received. In the phase ready-to-end, a process is forked to keep an objection raised until the flag gets set.

`class Environment` The two main customization mechanisms of UVM are the factory and the configuration database. The former allows to change at instantiation time the type of an object being created with a derived one; the latter allows a parent in the testbench topology to define properties for its children. Given that the build phase is called top-down, the parent can't directly set those properties in its build phase, as the layers below don't exist yet; instead, it leaves them packed in a configuration object in the database, so that children can retrieve it when it's their turn to be built.

The environment is fundamental for both. In its build phase, it retrieves a configuration object of type `env_cfg_t` from the database: most of the properties within it are put back in the database and passed down the hierarchy. Also, it instantiates the verification

```
constraint ab_dist_c {
  a dist { // skew a towards the boundaries
    0                        := 1,
    [1 : {NBIT{1'b1}} - 1]  :/ 1, // spread
    {NBIT{1'b1}}             := 1
  };
  b dist {
    0                        := 1,
    [1 : {NBIT{1'b1}} - 1]  :/ 1, // spread
    {NBIT{1'b1}}             := 1
  };
};
```

Listing 3.1: Weighted distribution constraint for the input operands of Intel's Pentium IV adder.

components through the factory, thus taking into account overrides requested by the test: for instance, `Printer` overridden by `BitBucket`, or `Coverage` by `StmCoverage`. Moreover, it takes care of connecting the analysis communication paths.

`virtual class BaseTest` According to the common approach presented in Section 1.1.2, this class takes care of duties common to all tests. Specifically, it configures the environment in a way that is suitable to most tests, leaving the callback `configure_env()` for them to perform additional tweaking. Considering that I wouldn't have needed tests to coordinate the execution of sequences on multiple agents, I avoided implementing a virtual sequence. Instead, the `BaseTest` instantiates a `TopSequence` object through the factory: this way, not only can I specify a different `TopSequence` class in each project namespace, but tests can also override it through the factory if needed.

Another fundamental job is to take care of processing command line plusargs:

- The file paths for the printer and scoreboard logs
- The flag `+quiet` to enable the `BitBucket` override
- `+n_txn` to choose the number of request transactions per test sequence to generate

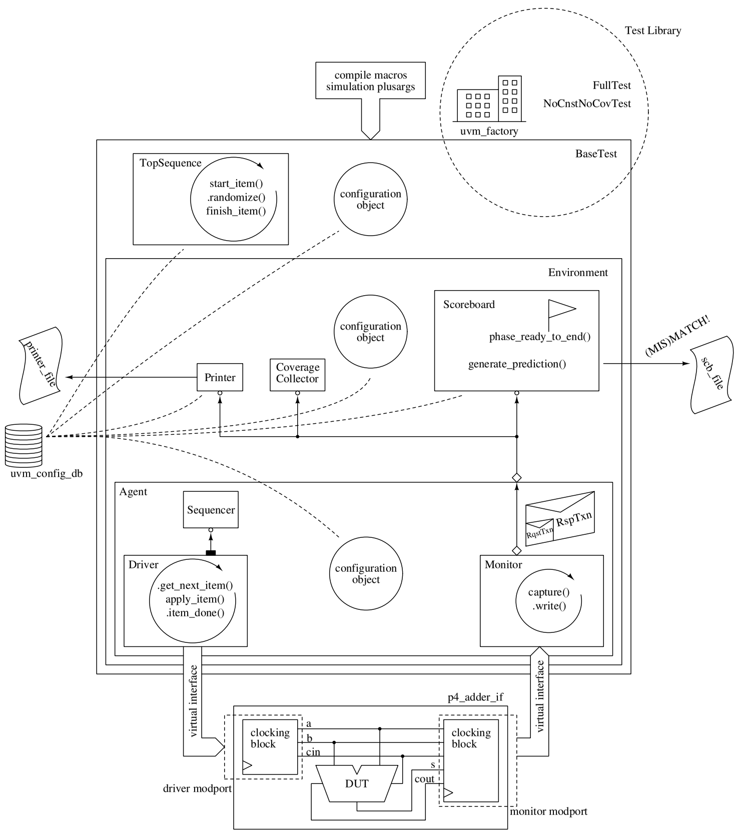
Also notice that when the verbosity is at least `UVM_HIGH`, for debugging purposes it prints the topology of the hierarchy, the content of the configuration database and a summary of the state of the factory.

3.2 Intel's Pentium IV Adder

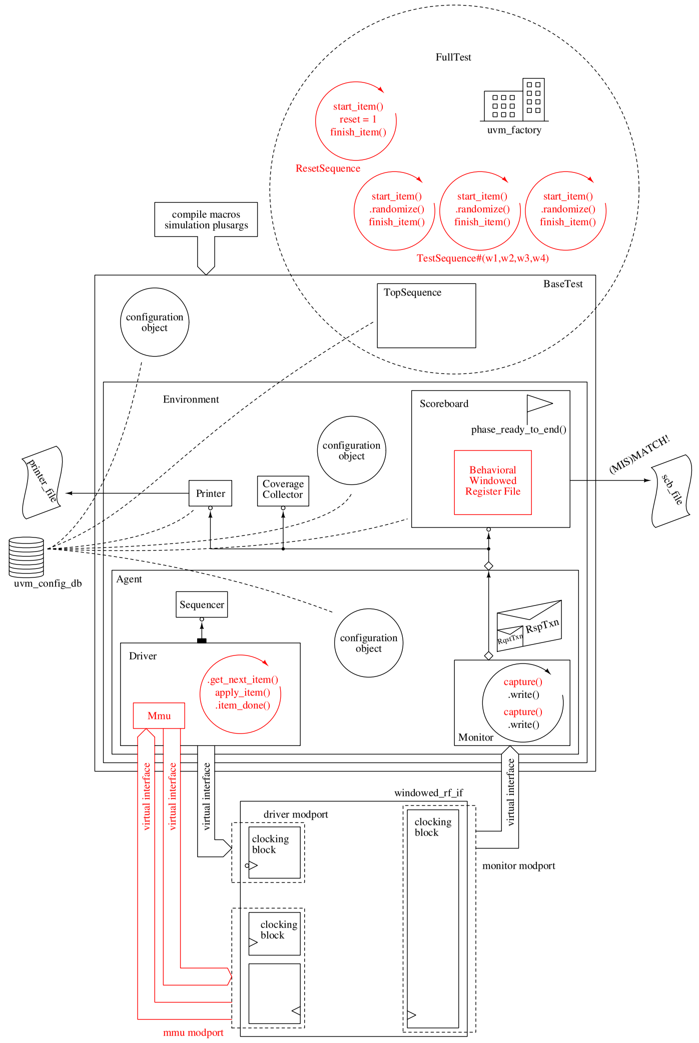
Starting from the interface, Fig. 4a shows that the interaction with the DUT is carried out through the driver and the monitor modports. Given that the circuit is combinational, I chose the following synchronization. The driver drives `a`, `b` and `cin` according to the clocking block defaults, namely 0 ns after the rising edge of the clock; then, `1step` before the subsequent rising edge, not only the monitor can sample the response of the adder, `s` and `cout`, but also the driven request, which is still valid.

As for the verification components tailored to the Pentium IV adder, the most interesting aspects are:

`class Agent` The driver and monitor virtual modport handles are retrieved in the build phase by the agent from its configuration object stored in the database. Only once the entire



(a) Intel's Pentium IV Adder.



(b) Fixed-Size Windowed Register File.
In red, the main differences compared to the adder testbench.

Figure 4: Conceptual representation of the UVM-based testbenches and their connection to the DUTs.

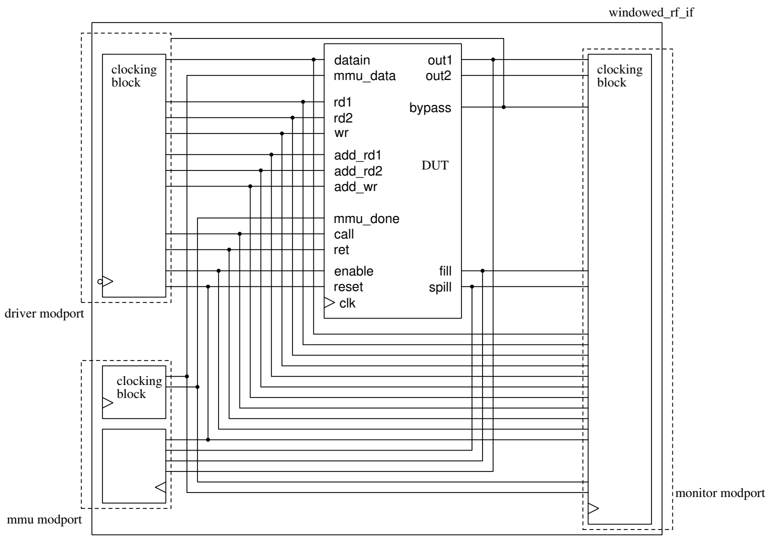


Figure 5: Detailed representation of the interface with the fixed-size windowed register file. The memory management unit has a dedicated modport through which it can respond to spill and fill requests: it samples the `reset` signal; during the spill operation it samples the `spill` and `out1` and drives `mmu_done`; during the fill operation it samples `fill` and drives both `mmu_data` and `mmu_done`.

hierarchy has been built, in the connect phase, the agent can use them to initialize the monitor and driver data members.

`class Scoreboard` It extends `BaseScoreboard` to specify the reference model and the implementation of the `generate_prediction()` callback. Given that the reference model is the addition operator inherent in the language, once having cloned the response transaction for its request fields the prediction boils down to a sum.

`class CnstRqstTxn` It extends `RqstTxn` to add the constraint in Listing 3.1 and skew the stimulus towards the boundaries of the representation interval, in an attempt to reduce the number of request transactions needed to cover the verification plan.

Making use of the techniques described earlier, two extended tests are available:

`class NoCnstNoCovTest` It's used to debug the active part of the testbench early in the development stages. It instantiates the environment as configured by `BaseTest`: the generated request transactions are fully unconstrained and there is no coverage collector.

`class FullTest` It overrides `RqstTxn` with `CnstRqstTxn` and configures the environment to instantiate a coverage collector, overridden with `StmCoverage`.

3.3 Fixed-Size Windowed Register File

Comparing the testbench architectures in Fig. 4, the first difference is in the interaction with the DUT. The windowed register file acts as a circular-buffer implementation for the top of the stack maintained in the main memory, thus I developed a verification component that emulates the

```
task apply_item(input RqstTxn rqst);

  /* synchronize on the driving active edge */
  @(vif.drv_cb);

  /* rf busy spilling/filling */
  if (!rqst.reset && vif.bypass) begin
    uvm_report_info("debug", "rf bypassed: waiting", UVM_HIGH);

    @(negedge vif.bypass);
    uvm_report_info("debug", "rf bypassed: fell", UVM_HIGH);

    @(vif.drv_cb);
  end

  vif.drv_cb.reset   <= rqst.reset;
  ...

endtask : apply_item
```

Listing 3.2: Snippet of the driver logic to translate incoming sequence items to signal-level. When the windowed register file is busy with a spill or fill operation, the translation is temporarily halted until the falling edge of the `bypass` signal notifies the end of the operation. Only a reset request is allowed to pass through during this time.

memory management unit, intending for it to be instantiated within the driver. The interface is detailed in Fig. 5. The driver communicates with the DUT through two separate modports:

- The driver modport is the one used to translate the incoming sequence items to signal-level. All DUT input signals, except `mmu_data` and `mmu_done`, are driven 0 ns after the falling edge of the clock, via a clocking block. This is because the windowed register file is active on the rising edge of the clock.

  The modport additionally includes an input for the driver: the `bypass` signal, which the driver uses to determine whether the windowed register file is currently occupied with a spill or fill operation. When such an operation is ongoing, the driver temporarily halts the translation of incoming sequence items until the register file becomes available again, which happens after the falling edge of the `bypass` signal. Only a reset request is allowed to pass through during this time, as it can abort the preceding fill or spill request. The SystemVerilog details are in Listing 3.2.

- The memory management unit has a dedicated modport, which bundles all the signals needed to respond to spill and fill requests. The inputs to the memory management unit are sampled `1step` before the rising edge of the clock, whereas the outputs, `mmu_data` and `mmu_done`, are driven 0 ns after it.

The monitor job is more complex due to the sequential nature of the DUT, which typically takes a clock cycle to respond, except when dealing with call requests that initiate a spill operation, in which case the response is delayed by an additional cycle. Both the request and the response are sampled on each rising edge of the clock, but the logic to distinguish between the two is within the monitor and is shown at high-level in Listing 3.3. In the interface, the monitor modport simply bundles all the DUT signals, both inputs and outputs, synchronizing their sampling on the rising edge of the clock through a clocking block.

```
task run_phase(uvm_phase phase);
  RspTxn rqst1, rqst2;

  /* skip init */
  @(vif.mon_cb);

  forever begin
    /* first edge */
    capture(rqst2, rqst1); // rsp to rqst2, rqst1
    if (rqst2 != null) begin
      ap.write(rqst2);
      uvm_report_info("capture", {"first edge: ", rqst2.convert2string()}, UVM_HIGH);
    end

    /* second edge */
    capture(rqst1, rqst2); // rsp to rqst1, rqst2
    if (rqst1 != null) begin
      ap.write(rqst1);
      uvm_report_info("capture", {"second edge: ", rqst1.convert2string()}, UVM_HIGH);
    end
  end
endtask : run_phase
```

Listing 3.3: Snippet of the monitor logic designed to manage the overlapped sampling on the rising edges of both the current request, which was driven on the preceding falling edge of the clock, and the pending response, whose request had been already sampled on the prior rising edge. The red arrow emphasizes that a request sampled on a rising edge is advanced to collect its response on the subsequent rising edge. The blue arrow underscores the same event, offset by one clock cycle.

As for the remaining verification components, the most interesting aspects are:

`class Mmu` The memory management unit extends `uvm_component` and implements the protocol described in Section 1.2 to respond to spill and fill requests. Conveniently, the model for the stack in main memory is already available in the language through the SystemVerilog queue data structure, which can be accessed as a stack through the `push_back()` and `pop_back()` methods.

To model the Finite State Machine (FSM), I chose a behavioral style similar to a SystemC `SC_THREAD` with a synchronous reset. In the run phase task, two processes are spawned:

- The logic that monitors the `spill` and `fill` signals, to detect and service spill and fill requests from the DUT.
- An alarm process, which wakes up at every rising edge and terminates when the `reset` signal is sampled high.

Given that the type of the parallel block is `join_any`, as soon as the alarm process terminates, the execution of the run phase task resumes. The survivor process is killed, the queue is cleared, and both processes are respawned. The key ideas are illustrated in Listing 3.4.

`class BehWindowedRf` Unlike the Pentium IV Adder, to extend `BaseScoreboard` and implement the `generate_prediction()` callback, I had to develop a behavioral reference model.

```
class Mmu extends uvm_component;

  /* behavioral stack in main memory */
  data_t mem[$];

  task run_phase(uvm_phase phase);

    /* fsm thread handler */
    process fsm_th;

    forever begin
      fork // separate fsm and synchronous reset

      begin : fsm_p

        fsm_th = process::self();

        forever begin
          @(vif.mmu_cb);

          if (vif.mmu_cb.spill) begin
            /* do spill */
          end

          else if (vif.mmu_cb.fill) begin
            /* do fill */
          end
        end
      end : fsm_p

      /* in case of a reset, restart the machine */
      begin : synch_reset_p

        do
          @(vif.mmu_cb);
        while (!vif.mmu_cb.reset);

      end : synch_reset_p

      join_any // the synchronous reset thread

      fsm_th.kill();
      fsm_th = null;
      mem.delete();
    end
  endtask : run_phase

endclass
```

Listing 3.4: Skeleton implementation of the memory management unit. In the run phase task, two processes are spawned: one containing the actual logic to respond to fill and spill requests, based on the signals sampled on the rising edges of the clock; the other is an alarm process devised to detect synchronous resets, kill the FSM, clear the stack in main memory and respawn both processes.

```
class CnstRqstTxn extends RqstTxn;

  /* call/return balancing:
   * count how many returns are possible */
  int unsigned can_return;
  byte unsigned ret_weight;

  function void post_randomize();

    /* balance */
    if (reset)
      can_return = 0;
    else begin // could be both call and ret, but call wins
      can_return += call;
      can_return -= (ret && !call);
    end
  endfunction : post_randomize

  constraint valid_ret_c {
    ret  dist {
      0 := (100-ret_weight),
      1 := (can_return ? ret_weight : 0)
    };
  }

endclass
```

Listing 3.5: Snippet of the adjustment of the distribution weight for the random bit `ret` to guarantee the validity of return requests. When the unsigned counter `can_return` reaches 0, the `ret` bit is restricted from assuming the value 1 until a subsequent successful call request occurs.

Working on the idea of the windowed register file as a circular buffer for the top of the stack in main memory, I decoupled these two aspects:

- The storage is implemented with a fixed-size array for the set of global registers and a SystemVerilog queue, accessed as a stack, for the windows. The elements contained in the queue are register subsets, the `subset_t` type, defined as a fixed-size array of registers that can act as *ins*, *locals*, or *outs*. The starting configuration is a queue containing 3 subsets, *ins* and *locals* of the active set, plus the *ins* of the adjacent set addressed as *outs*. Notice that to locate the current window, is sufficient to store the index of the active subset of *locals*. As a consequence:
  - Calling is equivalent to pushing a register set, the *locals* and the *outs* subsets of the newly activated window; thus the *outs* of the previously active windows become the *ins*, as soon as the index of the active *locals* is updated.
  - Vice versa, returning is equivalent to popping a register set.
- The circular management is achieved by storing the numbers of the most recent and oldest windows in the stack, traditionally called the *current* and *saved window pointers*. Counting in modulo the total number of windows, the condition:
  - Buffer full, which initiates a spill operation, is detected when a call request makes the pointers become equal.

– Buffer empty, which initiates a fill operation, is detected when a return request makes the pointers become equal.

`class TopSequence` Considering the higher complexity of the coverage model implemented in `StmCoverage`, in comparison to the case of the Pentium IV Adder, I chose to make `TopSequence` a hierarchical sequence object in an attempt to increase the total functional coverage within the same test. Internally, after executing the `ResetSequence` to establish the same known state for both the DUT and the reference model, the same parametric test sequence `TestSequence` is executed thrice, each time with parameters fine-tuned empirically to target different portions of the verification plan.

The template parameters of `TestSequence` serve to customize the randomization properties of the request transaction, which acts as a blueprint object. Notably, among the various constraints introduced by `CnstRqstTxn` when extending `RqstTxn`, the key constraint illustrated in Listing 3.5 ensures the validity of return requests. This is achieved by leveraging the `post_randomize()` function to dynamically adjust the distribution weight of the `ret` random bit. The implementation makes use of the unsigned counter `can_return`, initialized to 0. Following each randomization, the `post_randomize()` logic assesses whether the request has resulted in a valid call or return operation. In cases of valid calls, the counter is incremented; otherwise, it's decremented. Once the counter reaches zero, the weight assigned to the value 1 for the `ret` bit is dropped to 0. This effectively enforces a hard constraint that prevents return requests from being issued until the next call request.

4 Results

The results presented in the following are based on simulations performed using the commercial simulator `QueastaSim`[1]. The printer and scoreboard logs were then post-processed with custom *tcl* scripts, run within the simulator shell, and with MATLAB scripts.

4.1 Intel's Pentium IV Adder

The verification of the Pentium IV adder was carried out across various parameterizations, as outlined below:

```
for $i \in \{3, \ldots, 7\}$ do
    for $j \in \{2, \ldots, i-1\}$ do
        NBIT $\leftarrow 2^i$
        NBIT_PER_BLOCK $\leftarrow 2^j$
    end for
end for
```

In each scenario, the test sequence was configured to generate 100 request transactions and the seed was randomly selected by the simulator. In all cases, the Register-Transfer Level (RTL) code coverage analysis reported full coverage and the DUT passed the test with the following scoreboard output:

```
Scoreboard Summary:
  Transactions: 100 out of 100
  Errors      : 0
  Coverage    : 100.00%
```

[1]`QuestaSim-64, version 2020.4 linux_x86_64, Mentor Graphics Corporation`

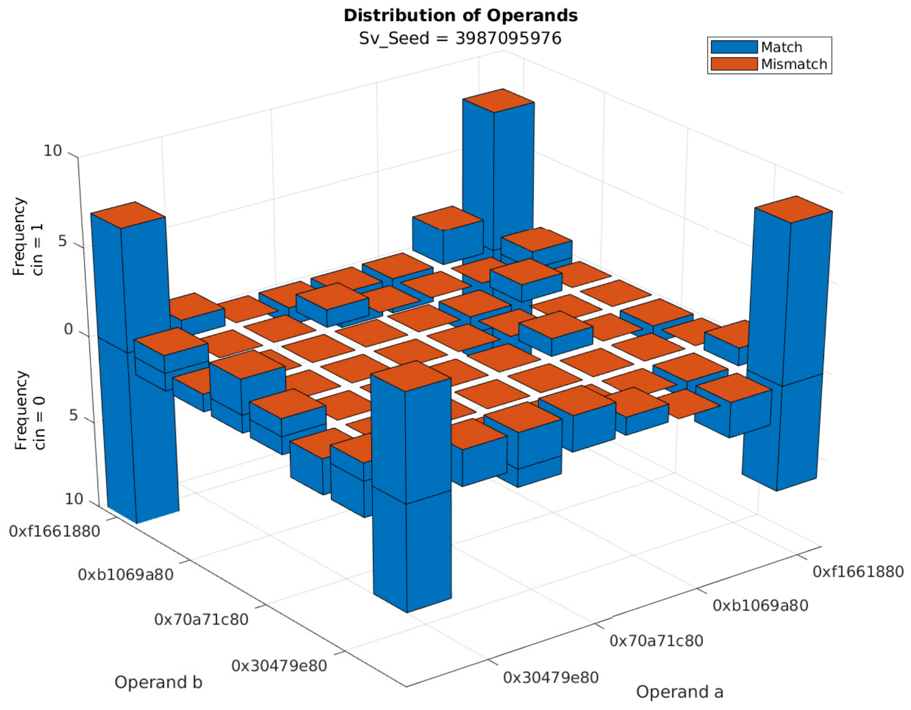


Figure 6: Visualization of the simulation logs for Intel's Pentium IV adder, parameterized with NBIT = 32 and NBIT_PER_BLOCK = 4; 100 request transactions. The bar plots for the cases `cin` = 0 and `cin` = 1 are joined along the $\hat{z}$ direction to emphasize that the weighted distribution constraint was effective in skewing the stimulus towards the boundaries of the support set. No violations occurred.

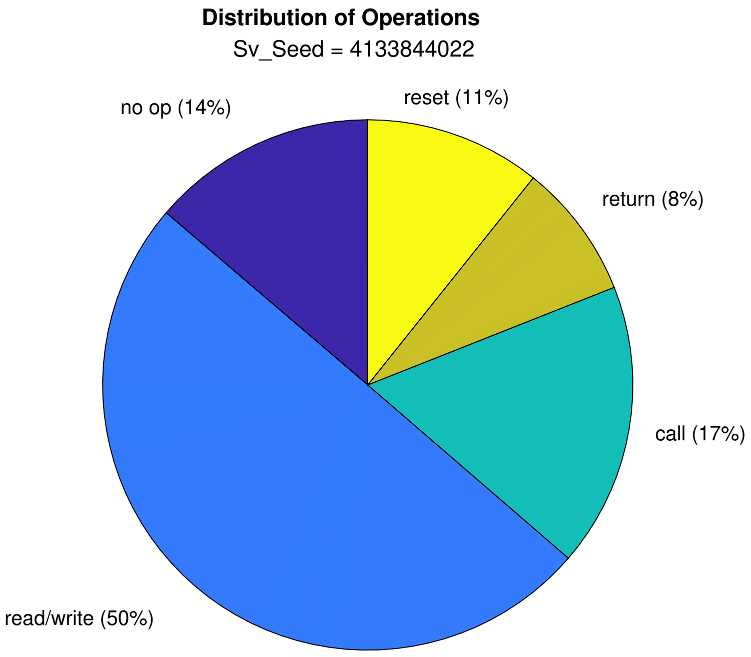
The post-processing of the logs, in the scenario where NBIT is 32 and NBIT_PER_BLOCK is 4, is in Fig. 6. This highlights how the weighted distribution constraint effectively skewed the stimulus towards the boundaries of the support set, a crucial aspect in covering the entire verification plan.

4.2 Fixed-Size Windowed Register File

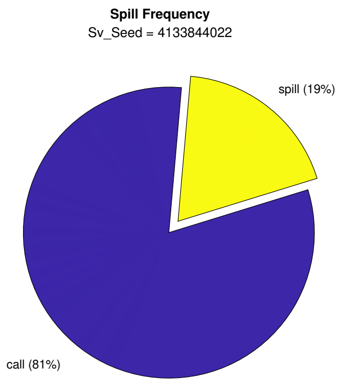
The verification of the fixed-size windowed register file was carried out across typical parameterization scenarios:

```
NBIT ∈ {8, 16, 32, 64}
NBIT_MEM ← 8                  ▷ memory management unit cycles depend on NBIT
NGLOBALS ∈ {8, 10, 12}
NLOCALS ∈ {8, 10, 12}
NWINDOWS ∈ {4, 8, 16, 32}     ▷ the smaller, the easier to trigger spill and fill
```

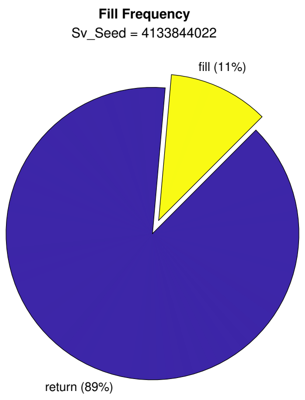
In all scenarios, using 100 request transactions per test sequence, totaling 300 transactions, resulted in achieving a total function coverage ranging between 74 % and 95 %. Increasing the



(a) The pie chart illustrates that all operations were executed. Read and write operations were the most frequent, accounting for 2995 transactions. Surprisingly, 14 % of the randomization attempts resulted in no operation being executed.



(b) Spill operations were triggered 196 times throughout the simulation.



(c) Fill operations were triggered 55 times throughout the simulation.

Figure 7: Visualization of the simulation logs for the fixed-size windowed register file, parameterized with NBIT = 32, NBIT_MEM = 8, NGLOBALS = 8, NLOCALS = 8, and NWINDOWS = 4; 2000 request transactions per test sequence, totaling 6000 transactions. No violations occurred.

number of request transactions per test sequence to 2000 proved sufficient to achieve a coverage of 100 % in most cases, while some scenarios required up to 5000 transactions per test sequence. To further stress both the testbench and the DUT, the number of request transactions was increased to 500 000 per test sequence.

The DUT passed the test in all scenarios, but the RTL code coverage analysis unearthed undesired redundancy in the `transition_logic_p` of the FSM that controls spill and fill operations. As shown in the following excerpt, the enumerated cases already cover all the enumeration constants, and the `others` case prevents the correct interpretation of states and transitions during `QueastaSim` compilation.

```
type state_t is (RUN, CWP_DOWN, SPILL_ONE, WAIT_SPILL, CWP_UP, WAIT_FILL, FILL_ONE);
...
transition_logic_p : process (...) is
begin

  case state is
    when RUN        => ...
    when CWP_DOWN   => ...
    when SPILL_ONE  => ...
    when WAIT_SPILL => ...
    when CWP_UP     => ...
    when WAIT_FILL  => ...
    when FILL_ONE   => ...

    when others     =>
      next_state <= RUN;

  end case;

end process;
```

The post-processing of the logs, in the scenario where NBIT = 32, NBIT_MEM = 8, NGLOBALS = 8, NLOCALS = 8, and NWINDOWS = 4, is in Fig. 7. This highlights the distribution of the executed operations, with an emphasis on call and return requests that triggered spill and fill operations.

5 Conclusion

In this analysis, I kept expanding my knowledge in SystemVerilog while focusing on mastering the UVM framework to structure reusable, scalable, and efficient testbenches for the functional verification of digital designs. Compared to the development process of the previous introductory step, the adoption of the UVM resulted in faster and higher quality development. I was able to take advantage of an extensively tested code base, directing the time and effort saved towards the most challenging problems of the designs I had to verify.

The hands-on experience encompassed the verification of two diverse designs: Intel's Pentium IV adder and a fixed-size windowed register file. Especially with the increased complexity of the latter design, both SystemVerilog and the UVM truly shined. Reflecting on the testbench I had developed in the *Microelectronic Systems* course laboratories, I once again recognize the limitations of the directed testing approach. The test program consisted of VHDL procedures implementing the recursive solution to the *Tower of Hanoi* puzzle at the signal level. It proved to be very error-prone to develop and it only targeted a subset of the DUT features. In contrast, leveraging the high-level features of SystemVerilog within the UVM framework, I could simplify the modeling of the memory management unit, exercise the DUT with constrained-random stimuli, and compare its responses with those of a behavioral reference model. All of this, while tracking progress through both code and functional coverage.

In summary, this endeavor underscores the paramount importance of acquiring expertise in advanced verification methodologies. The SystemVerilog language and the UVM combined confer invaluable skills to tackle the challenges of complex digital designs.

References

[1] SPARC International, *The SPARC Architecture Manual: Version 8*. Prentice Hall, 1992, ISBN: 9780138250010 (cit. on p. 8).

[2] "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.

[3] "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.

[4] 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. 4).

[5] R. Salemi, *The UVM Primer: A Step-By-Step Introduction to the Universal Verification Methodology*. Boston Light Press, 2013, ISBN: 9780974164939.

[6] "IEEE Standard for Universal Verification Methodology Language Reference Manual," *IEEE Std 1800.2-2020*, pp. 1–458, 2020. DOI: 10.1109/IEEESTD.2020.9195920 (cit. on p. 4).

Acronyms

DUT	Device Under Test
RTL	Register-Transfer Level
UVM	Universal Verification Methodology
ASM	Algorithmic State Machine
FSM	Finite State Machine