NTNU – Trondheim
Norwegian University of
Science and Technology

UVM Verification Framework

Mads Bergan Roligheten

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Vitaly Marchuk, Atmel Norway AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem description

Atmel has a wide range of IP designs and a good, reusable and efficient verification framework is extremely important to have short time to market. UVM (Universal Verification Methodology) is a methodology for functional verification using SystemVerilog, which is a set of standardized libraries of SystemVerilog. The student will investigate how UVM can be used to build a reusable verification framework. He has to take decisions on the following tasks:

- Synchronization between transaction level and RTL design;
- Packing and unpacking transactions and driving them to the design under test;
- Configuration database and reusability;
- Constrained randomization;
- Functional coverage and test execution control;

The verification framework can be used on any open source RTL. This is interesting and challenging work on top edge of industry verification. It requires knowledge of SystemVerilog, object-oriented programming and digital systems.

Assignment given: January 2014

Supervisor: Kjetil Svarstad, IET

Assignment proposer / Co-supervisor: Vitaly Marchuk, Atmel Norway AS

Abstract

The importance of verification is increasing with the size of hardware designs, and reducing the effort required for is necessary to increase productivity. This thesis covers the creation of a reusable verification framework for processor verification using the Universal Verification Methodology (UVM). The framework is used to verify three simple processor designs to evaluate its potential for reuse. The three processors include a synchronous, asynchronous and a stack based processor. A pure UVM implementation is evaluated against the use of external checking by Assertion Based Verification (ABV), which is found to provide a better overview. The framework is shown to be highly reusable, especially for input generation, and can be used for both synchronous and asynchronous design. The high reusability is a key part of increasing productivity gained by removal of redundant work. This framework is intended as a proof of concept, and is does not provide a complete verification for each of the designs.

Sammendrag

Behovet for verifisering øker med størrelsen på hardware design, og det å redusere innsatsen som kreves, er nødvendig for å øke produktiviteten. Denne oppgaven dekker oppsett av et gjenbrukbart rammeverk for prosessor verifisering ved hjelp av Universal Verification Methodology (UVM). Rammeverket blir brukt til å verifisere tre enkle prosessor design for å evaluere potensialet for gjenbruk. De tre prosessorene inkluderer en synkron, asynkron og en stack basert prosessor. En ren UVM implementasjon blir vurdert opp mot bruken av ekstern sjekking ved hjelp av Assertion Based Verification (ABV), som viser seg å gi en større grad av oversikt. Rammeverket blir vist til å være svært gjenbrukbart, spesielt for generering av inngangssignaler, og kan brukes med både synkrone og asynkrone design. Den økte graden av gjenbruk er et nøkkelelement mot økt produktivitet, som oppnås ved å fjerne øverflødig arbeid. Rammeverket er ment som et proof of concept, og inkluderer ikke en fullstendig verifisering av de enkelte prosessor implementasjonenen.

Preface

This thesis concludes a 5-year Master programme at the Norwegian University of Science and Technology (NTNU), under the department of Electronics and Telecommunication (IET). The work was carried out at Atmel Norway AS, under supervision of Vitaly Marchuk, who proposed the topic.

I would like to thank my supervisor at NTNU, Professor Kjetil Svarstad for his guidance, and I would like to give special thanks to Vitaly for his great support and motivation throughout the project period.

Contents

Chapter 1

Introduction

The development of integrated circuits is prone to include some errors in the design, especially considering the increased complexity of newer generation technology. Verification is an important part of the hardware development process, seeking to uncover as many bugs as possible before the design is sent to production. With the increase of design productivity the verification constitutes an increasing amound of the effort spent on each project. The Wilson Research Group study of 2012 show that time spent on verification had increased by 15 the five previous years, and averaged at 56% [5]. It also shows that the average number of verification engineers vs design engineers has increased 75% in the same period, peaking at a one-to-one ratio, and that design engineers actually spend a significant amount of time doing verification as well [5]. This shows that despite advances in verification methods, there is still a need to focus on improving and reducing the time spent on this.

One of the reasons for increase in design productivity is the increased amount of reusable model libraries that simplifies the most common tasks. A key focus for improving verification would be to enable significant reuse of the verification methods. Not only the reuse of common modules, but complete verification environments as well, could significantly reduce the effort required for each design cycle. This motivates the creation of an efficient and reusable framework for verification.

The process of testing a design has evolved from simple directed testing, to elaborate verification methodologies alongside the increase in design productivity. Though simple directed tests are fast and easy to write for small designs, satisfactory verification requires significantly more elaborate systems. Several languages emerged in response to this need, providing extensive libraries to aid in this work. Two of the most known are *e*[2] and SystemVerilog[4], spawning several different methodologies in different circles. The Universal Verification Methodology (UVM) [12] is the result of an attempt to combine the different methodologies and create a best practice to ease interoperability between engineers.

For this thesis three different RISC processor designs have been chosen to provide similar but challengingly different systems to verify. These are the synchronous [6], asynchronous [17] and stack [16] processors. The synchronous design has a rather generic implementation providing a good starting point to build a framework. The asynchronous version adds the complications of removing the system clock, and finally the stack processor features a different processing concept. Creating a framework able to verify these different designs with minimal effort will be able to show the benefit and potential of such a system.

It is necessary to specify that the focus of this thesis is IP level designs, making some of the UVM features less efficient as it largely intended for System on Chip implementations with focus on transaction level communication between modules. The thesis will show that the UVM can be just as useful for single IP designs as well by utilizing the features that promote reusability.

The UVM will be utilized for building the verification framework, while at the same time addressing what features of the methodology might not be as useful, or even inefficient. This discussion is brought up in [10], where the authors analyze the cost-benefit of the UVM macros. Furthermore a discussion can be made of whether the framework should be restricted to only UVM components or if some tasks are best handled by independent modules. An alternative to the use of UVM checkers and registry models is by means of assertions. Assertion Based Verification (ABV)[13] provide a direct way of specifying functional behavior and will be evaluated as well as looking into ways that this can be made more reusable.

The thesis will feature a qualitative study of a UVM framework and focus on showing the potential of a reusable framework as well as highlighting the strengths and weaknesses of the methodology. It might also aid in further encouraging the adaption of this standard, or as stated in [7] “Selling UVM to hesitant clients. Bring them kicking and screaming into the 21st century”

The rest of the thesis is partitioned as follows:

- Chapter 2 will provide some background on the topic of verification and some more information on the languages and methodologies.
- Chapter 3 covers the creation of a pure UVM framework with a detailed description of the various parts.
- Chapter 4 will discuss alternatives and potential improvements.
- Chapter 5 describes the changes required to adapt the framework to the other processor designs.
- Chapter 6 evaluates the performance of the framework.
- Chapter 7 concludes the thesis and suggests possible future work.

Chapter 2

Background information and theory

To understand the need for building a verification framework we need to look into how the methods for testing have developed over time. Before looking at the development of different methodologies, let's first examine the concept of verification.

2.1 Verification

What do we mean when we talk about verification?

Verification is simply the process of making sure the design does what it is supposed to do according to a specification. It has essentially two tasks; providing input stimulus that is able to control the design into all possible states, and be able to observe how the design responds.

Verification is not just testing, but testing to the extent that you can be confident of the correctness of a design. However, there is no way to completely verify a design, except for really small designs, because of the increasingly large number of possibilities. Thus we require methods that can provide as good as possible confidence that the verification is satisfactory. Verification is commonly done through simulation where you either create a model that behaves according to the specification for comparison, or specify directly the expected states/values at a given time.

2.1.1 Directed test

Directed testing is a quick and easy way to write simple tests for new designs and is often used during the initial stages to test specific cases. It requires little to no overhead and allows for direct targeting of expected signal transitions. For small designs this is an efficient way to write basic tests for early checking of correctness, but with increasing complexity and the need for exhaustive testing to confidently verify a design, this method requires significantly more effort to write and quickly becomes a tedious task. While it is still possible to create a solid set of tests this way, it usually requires significant effort to setup and maintain the code. It is also hard to reuse any part of a test being written for a specific design. Another major limitation is the human factor, meaning that we are only able to test the limited set of cases that is conceivable. Though clever minds can think of the most likely problems and take special conditions into account, there may still be a huge set of unknown combinations potentially creating bugs that would be difficult to imagine.

Together with the rapid increase in design efficiency it soon became necessary to find better methods for testing and verification. This is what led to the creation of dedicated verification languages and more structured methodologies. It also became necessary with dedicated verification engineers to handle this part of the process.

2.1.2 Hardware Verification Languages

To meet the needs of verification engineers several HVLs was created, the most widely known of these being *e* [2], Vera/OpenVera [15], SystemC [3] and SystemVerilog [4]. Domain-specific languages like *e* and Vera provided several efficient verification features later improved with OpenVera and assertion based verification.

SystemC using the C++ library provide an all domain language and is a general-purpose programming approach to hardware description and verification, and serves as a contrast to SystemVerilog that is based on a HVL [11].

2.1.3 SystemVerilog

SystemVerilog is one of Accelleras efforts to combine all the domains of both hardware description and verification into a single language. The domains being

Netlist, Register transfer, General programming, Testbench, Temporal properties and Functional coverage[11] had previously been separated by domain specific languages, limiting the interoperability between engineers. In 2005 SystemVerilog was adopted as an IEEE Standard [4], extending the design features of Verilog[1], and adding advanced verification features based on OpenVera.

The following is a description of the most important advantages SystemVerilog features for verification.

Classes
The ability to separate and organize functionality in classes enables increased reusability and control of the verification environment.

Constraint-random
A good verification language provides the possibility of randomizing variables as a way to remove the human factor. Randomized inputs can reach unthinkable states in the design, and together with constraints excluding illegal values, more of the design can be explored in a shorter amount of time. Randomization can also be used without constraints to check proper error handling of faulty input.

Communication
Efficient methods for communication between classes and the RTL net-list are necessary, and are enabled through interfaces and modports. The interface bundles the I/O connections and allow direct access through different levels of the hierarchy, reducing the common issue of spaghetti code. Interfaces can also implement the necessary functionality for bus transfer protocols, enabling efficient bus communication. Efficient communication is also the basis for Transaction level Modeling.

Synchronization
To be able to use TLM together with RTL code, it requires methods for synchronization between the threaded transaction level description and the RTL signals. This is added by the use of mailboxes for messaging and semaphores to control execution order and access to resources.

Assertions
Assertions provide methods for checking temporal and functional properties and can be added several places in the code for immediate or delayed checking of properties. More on assertions in chapter 2.1.6

Functional coverage
With the use of coverpoints and covergroups it is easier to track the progress

and monitor exactly what has been tested, and it is especially necessary when randomizing the test variables.

Along with several improved features to the design part of the language, SystemVerilog looked to be not only the most complete HDL, but was shown to perform better in terms of execution speed for hardware simulation compared to VHDL and SystemC [9]. However, the use of the many features was not necessarily easy, and many struggled to use these efficiently. The lack of a shared understanding of best practice and with no standardized environments, it was hard to share work between engineers, limiting cooperation and reuse [8].

2.1.4 Methodologies

Even before SystemVerilog, with multiple languages in use, different vendors created their own methodologies to help increase productivity in verifying designs. The methodologies were class-based, leaving the user with only small portions of code to modify, and had a high focus on reusability.

The first focus on reuse was the *e* Reuse Methodology (*e*RM) from Verisity Design. Cadence Design Systems later combined this with SystemVerilog to create the Unified Reuse Methodology (URM). From Synopsis came the Reference Verification Methodology (RVM) based on OpenVera, which with OpenVera being used as a base for SystemVerilog verification, later became the Verification Methodology Manual (VMM). Mentor created the Advanced Verification Methodology (AVM) based on a combination of SV and SystemC, and this was used together with URM in the creation of the Open Verification Methodology (OVM).

In 2011 Accellera approved version 1.0 of UVM as the result of an effort to unify the different methodologies. The idea was to create a best practice that relies on strong, proven industry standards. Figure 2.1 show the relations between the different methodologies and how they have led to UVM.

The most recent implementation is UVM 1.1d [12], and is the one used in this thesis.

2.1.5 UVM

Building upon the already extensive library of SystemVerilog, UVM adds a comprehensive library of classes to help increase standardization and interoperability.

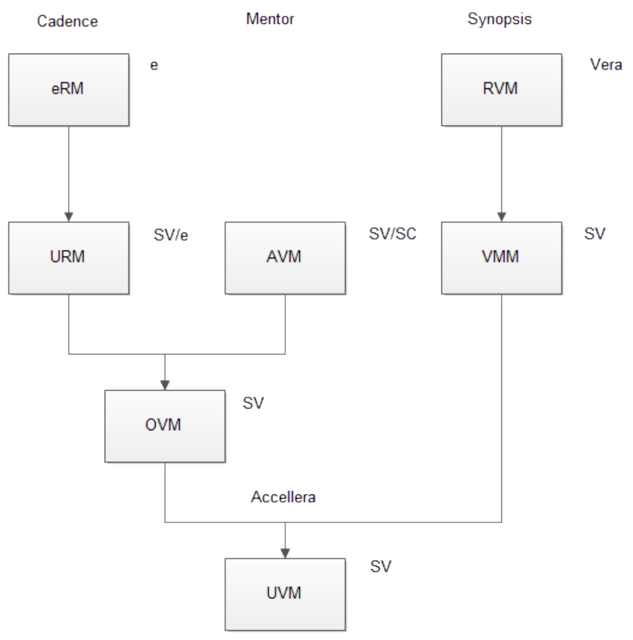


FIGURE 2.1: Development of methodologies

The library is coded entirely in SystemVerilog source code[8], enabling not only a highly standardized structure, but also provides the freedom to choose the most appropriate tools for the given task. This makes the UVM a powerful toolkit for verification.

Some of the most important advantages of using UVM are described below.

Constraint random
UVM makes efficient use of the constrain-random functionality enabled by SystemVerilog, and builds its transactions based on this.

Single class hierarchy
The classes in UVM are all expanded from a single root class, '*uvm_object*', enabling key functionality to be available throughout all the components.

Object factory
The factory is the central mechanism for creating objects or components. By registering objects with the factory, we enable a fully configurable hierarchy that can be modified with specialized implementations at run-time.

Configuration/resource database
One of the most important facilities of UVM is a resource database that makes

configurations globally accessible. This allows test specific configurations to be added to the testbench independently.

Phases
The different class threads running in a UVM environment are coordinated using a phase based execution, ensuring proper ordering of events.

Prewritten code
As all the classes used in a UVM environment are predefined with significant functionality, there is less work left to the user for each implementation.

This is just a brief overview of the advantages, as more detailed information will be discussed when implementing the different classes in Chapter 3

2.1.6 Assertion based verification

Assertions is a verification method that provide a good way of targeting specific behavior. This is done by separately describing expected transitions or results for each property we want to verify. System Verilog Assertions (SVA)[4] is a standardized assertion language as a part of SystemVerilog and provide a well defined system for assertions.

Assertions are used to describe temporal and functional properties of the design, stating when and whether this should hold true. These properties are monitored during simulation and provide valuable feedback when verifying a design.

There are several ways to utilize assertions, both for monitoring that the simulation is executed properly and that the design responds properly. Assertions can also be added both at RTL-level and TLM-level [14].

2.2 Designs under verification

The following is a short description of each of the three designs that are targeted for verification.

2.2.1 Synchronous processor

The synchronous processor used here is a simple 16-bit RISC CPU featuring a limited instruction set [6]. The processor design has not been thoroughly tested and may contain some bugs, making this a good starting point for both developing and testing the framework.

2.2.2 Asynchronous processor

The synchronous processor is also being used as the base for another thesis, which set out to convert this simple processor to asynchronous logic [17]. Verification of asynchronous logic may generate difficulty regarding timing and synchronization, but as it is based on the same instruction set, the transition should be less difficult. The main issue this version presents, is the timing of when to drive input data, and knowing when the results can be checked. The work on this adaptation is being carried out simultaneously, thus providing a real-world approach for the verification process.

2.2.3 Stack processor

Another concurrent thesis aims to implement a stack based processor for energy harvesting systems [16]. This provides another processor design for verification, with added challenges due to the completely different design principle. It features an even more limited instruction set, but the new instructions together with the different structure should provide additional challenge when adapting the framework to this design.

Chapter 3

Building a framework

To be able to verify three different designs efficiently, we need a solid framework that ensures minimal effort when changing the Design Under Verification (DUV). For this to be possible, it requires identification of reusable parts and making these as generic as possible. The process can be divided into three steps as we have three different designs to work with.

- The synchronous processor features a fairly simple instruction set and common architecture, making it a good starting point for building a generic framework.
- The next step is then to adapt this to the asynchronous version of the processor. Because it uses the same instruction set this transition should not be too difficult.
- Lastly, by adapting to the stack processor, featuring a completely different instruction set as well as completely different architecture, it will be possible to show the full potential of the framework and whether or not this is a feasible system.

3.1 Verification parts

Before discussing the details of the implementation, let's take a look at what is necessary to create a constrained-random verification scheme.

1. Input has to be generated in a randomized fashion.
2. Behavior of the design must be checked for functional correctness.
3. Monitoring the process to keep track of what has actually been tested.

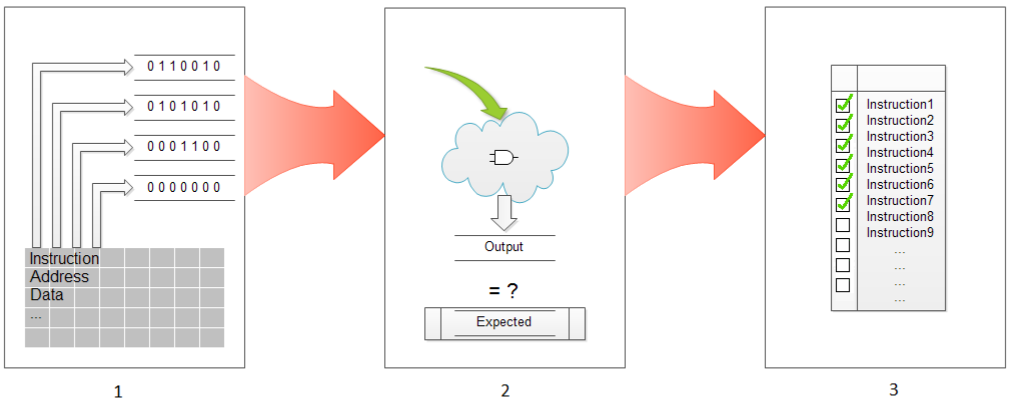


FIGURE 3.1: The 3 parts of verification

1. Input generation

Planning the test inputs is a big part of the effort invested when writing directed tests as you have to specify every situation you want to check, and it also contains only a check of predicted bugs. How the input is generated greatly affect both the effort required and the range of situations we are able to check. Constrained-random generation is the key feature that not only reduces the amount of time spent on this part of the process, but also enables the finding of unpredicted bugs. This reduces the tedious work of planning input data to only having to specify the variables and add some constraints removing illegal values or limiting the range of the test.

2. Functional checking

After generating proper input, the next step is to confirm that the design behaves as intended. Functional checking can be done in several ways, but is essentially a case of comparing the designs response with the expected result based on the specifications. How this expected result is determined depends on the design description and what tools are in place. Two methods for functional checking will be explored later debating whether to use a pure transaction level UVM implementation or add independent parts to the system.

3. Coverage collection

Verification is a never ending race to test as much as possible is as little time as possible, but it is near impossible to completely verify a design of considerable size. With the use of random input generation it is essential to collect information of what is actually verified. Coverage is a measure of how thoroughly a design has been tested based on predefined goals of what is considered sufficient.

Here we have to make a distinction between code coverage and functional coverage. Code coverage is a feature provided by simulation tools that keeps track of the lines of code that has been visited during simulation. This provides information on whether all states are reachable or if the test is thorough enough to reach all states.

Functional coverage is specified directly using coverpoints and covergroups, allowing engineers to create a boundary of what is the minimum acceptable coverage. Covergroups are categories of events that are sampled when the specific event occurs. Coverpoints decides what parameters of the event should be counted toward total coverage. An example of this is shown in Section 3.5

3.2 UVM structure

From the previous chapter, we know that UVM was aimed at creating a new standard by combining the best practices from the previous generations. Throughout this chapter the advantages will not only be shown in use, but also evaluated in terms of their actual value for reusability and increased productivity.

In order to determine the full potential of the UVM this chapter will cover the implementation of a pure transaction level UVM environment. This will involve an in-depth description as each of the components will be evaluated in terms of efficiency, reusability and their overall benefit to the system.

An illustration of the framework implementation is shown in Figure 3.2 giving a picture of how the structure is set up, and providing a reference to relate the different components described later on.

The advantage of the single-class hierarchy is that each of these components requires very little effort to set up. Because of the standardized structure all components are created in a strict fashion for registration with the object factory to enable proper timing and communication.

3.2.1 Macros

Before going into details of the implementation, there are some features that require special attention. These features are made available through macros, and range from essential utility macros to optional functionality like copy, compare and pack. Though the use of macros usually simplifies code writing and is necessary

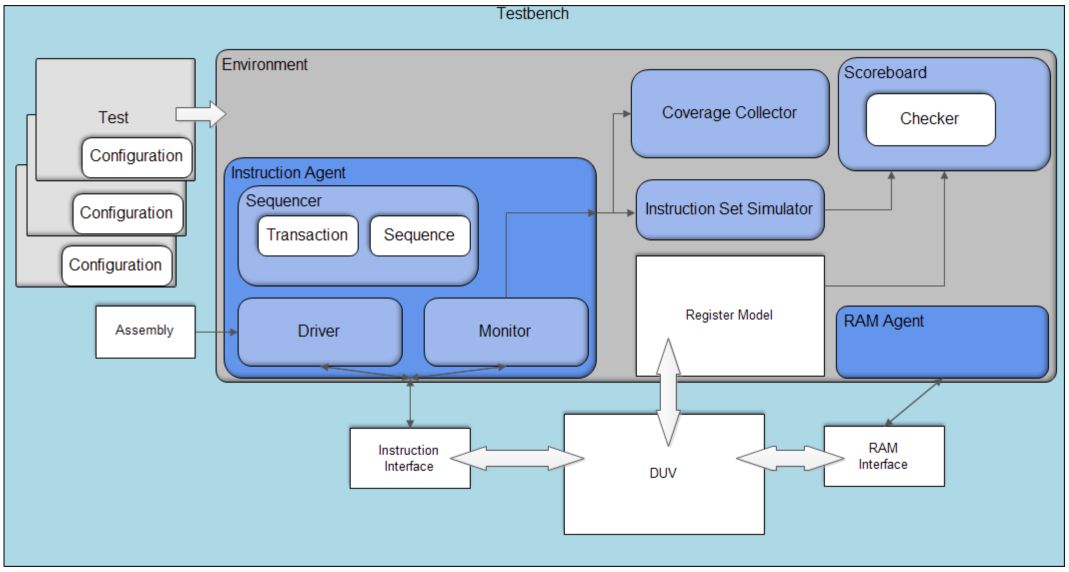


FIGURE 3.2: Implemented framework in UVM

to increase reusability, a concern voiced in [10] indicate that some of these may be inefficient. A cost-benefit analysis is performed to show the real benefit and the recommendations for the relevant macros are repeated here.

'uvm_object|component_utils is the essential macro that registers the class with UVM factory and should always be used to ensure proper type registration.

'uvm_info|warning|error|fatal provides a reporting mechanism that enables messages like error reporting and debug information to be filtered based on the verbosity. These macros are highly efficient and should be the primary choice for this purpose.

'uvm_field_* macros are used to manipulate the transactions throughout the environment with functions like copy, compare and pack/unpack. Though these macros have been significantly improved from OVM to UVM, it is still suggested to avoid these and instead replace with custom implementations for the necessary functionality [10]. The need for these in the framework implementation will be discussed in Section 3.3.1.

'uvm_do_* is a simplification of the execution of sequences and is not recommended to use [10], as it essentially adds a large set of additional execution phases. These macros are intended to hide some execution steps, but do this at the cost of significant code bloat. The same functionality can be rather simply added directly, as shown in Section 3.3.2, providing better control of the process.

Other macros are evaluated in the paper, but are not used in this implementation.

The main reason to avoid the inefficient macros become clear when considering reusability, as inefficient code does not provide a good base for reuse.

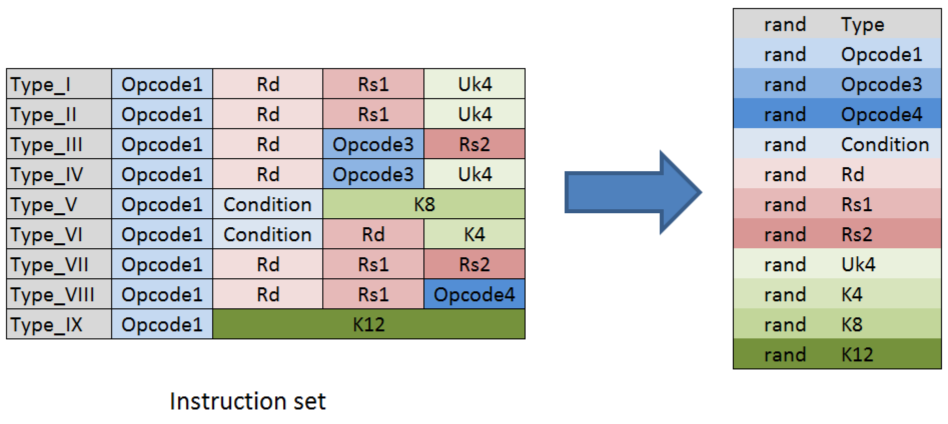
3.3 Input generation in UVM

The necessary classes for input generation are the ones contained in the instruction agent, specifically transaction, sequence and driver.

3.3.1 Transaction object

The core component of a UVM environment is the transaction object. This is the foundation of data transfer within the UVM environment as it defines the data types used for communication. It serves as a container for the set of variables required for a specific type of transaction. For a bus transaction this would typically be data, address and request/grant variables, while other transactions can have more or fewer variables depending on its purpose.

In the case of a processor design the transaction will contain variables required to create instruction vectors, as the main control inputs are the instruction word and the clock. The key of constraint random testing is to efficiently randomize the input instructions while being able to limit or control the types or ranges of instructions during runtime. This is done by extracting all the variables specified in the instruction set documentation [6] as shown in Figure 3.3.



Type_I	Opcode1	Rd	Rs1	Uk4
Type_II	Opcode1	Rd	Rs1	Uk4
Type_III	Opcode1	Rd	Opcode3	Rs2
Type_IV	Opcode1	Rd	Opcode3	Uk4
Type_V	Opcode1	Condition	K8	
Type_VI	Opcode1	Condition	Rd	K4
Type_VII	Opcode1	Rd	Rs1	Rs2
Type_VIII	Opcode1	Rd	Rs1	Opcode4
Type_IX	Opcode1	K12		

Instruction set

rand	Type
rand	Opcode1
rand	Opcode3
rand	Opcode4
rand	Condition
rand	Rd
rand	Rs1
rand	Rs2
rand	Uk4
rand	K4
rand	K8
rand	K12

FIGURE 3.3: Extracting variables for randomization

An advantage that can be utilized here is the enumeration of these variables to increase both readability and reusability. By naming all the valid values for each of these variables it is possible to put together random instructions that are more easily monitored and managed in the test. The benefit of this is best shown in Figure 3.5 in Section 3.3.2 where it is much easier to keep track of and know what instruction was executed. Also, as with any variables that may be referenced more than once, this helps limit the places to edit when making changes.

```
rand instruction_type iType; // Enumerated instruction types
rand opcode1 opc1; // Enumerated opcodes
{...}

  // Register field macros
  ‘uvm_object_utils_begin(utran_inst)
    ‘uvm_field_enum(instruction_type, iType, UVM_ALL_ON)
    ‘uvm_field_enum(opcode1, opc1, UVM_ALL_ON)
    {...}
  ‘uvm_object_utils_end

constraint idist{ iType dist { Type_I := 3, Type_II := 4, {...}
}

constraint isnttype{ iType == Type_I -> opc1 inside {ori4_op1, add2_op1, sub2_op1};
                  iType == Type_II -> opc1 inside {lw_op1, lb_op1, sw_op1, sb_op1};
                  {...}
 }
```

The code snippet above shows an example of the transaction object for instructions. The constraints are essential to ensure generation of valid instructions and are accompanied by a priority list to ensure equal distribution of instructions. The values can later be overwritten either in a sequence or using the configuration database enabling active adjustment of the input generation.

We notice here that *'uvm_field_** is enabled for each of the variables, going against the recommendation discussed earlier. The reason for this is that the field registration not only enables the macros, but also enables the variables to be visible outside the transaction. Without this enabled they would not be listed in Figure 3.5.

The predefined macro functions are not utilized here, but should they be necessary, then the recommendation should be followed by overwriting with custom functions. This does require some additional effort when creating the transaction, but is of benefit in the big picture considering reusability.

3.3.2 Sequence

A sequence takes care of the creation, randomization and finalization of transaction objects making them ready to be used as simulation inputs. The sequence uses a transaction object and creates multiple iterations of the object to form a stream of input data. This can create complex sequences that simulate operations like bus read/write protocols, and is another step up in abstraction as we no longer have to deal with the details of each transaction.

It is also possible to create virtual sequences contain several of the normal sequences. This enables even better control of the execution in an environment that deals with several verification components. The feature has not been implemented in this framework due to lack of necessity, but is mentioned as a possible future improvement

```
task body;
  // Create a transaction of the given type
  m_utran_inst = utran_inst ::type_id::create("m_utran_inst", null);
  forever
  begin
    start_item(m_utran_inst); // Synchronize with sequencer / wait for request
    assert( m_utran_inst.randomize() ); // Randomize transaction variables
      else 'uvm_error(...);
    finish_item(m_utran_inst); // Give completed item to sequencer
  end
endtask
```

An example of a standard sequence implementation that was used to verify the three CPUs is shown above. The four lines executed in the body task, replace the *'uvm_do* macro described earlier with the essential functions for sequence operation. Though the macro adds this to a single function call, the additional functionality provided by the macro are rarely necessary and add a set of execution phases that only slow down the simulation.

The assert statement added before randomization is used to ensure that proper randomized values are generated. The randomization may fail if there are conflicting constraints, and should be checked to prevent unexpected values in the simulation. An error can then be reported using the reporting mechanism.

Even though we could generate complex sequences, this is not necessary for general simulation of the CPUs as it only requires the creation of single random instruction for each clock cycle, and thus requires no special implementation.

However, should you require some special functionality this can easily be added, and as an example, another sequence was created to initialize the processor's register. The initialization sequence simply forces the variables in the transaction to have a specific value so that we control which instruction is generated. This is done using ".randomize() with" allowing direct control over the randomized values. The code snippet below shows what little change is required to create more specific sequences. These changes can range from limiting one variable to being able to directly control all the fields.

The same technique can also be utilized to reach corner cases in the simulation by creating specialized sequences.

```
task body;
  m_utran_inst = utran_inst ::type_id::create("m_utran_inst", null);
  start_item(m_utran_inst);
  assert( m_utran_inst.randomize()
    with {m_utran_inst.opc1 == ldir1_op1 &&  // Set variable value and exclude it
           m_utran_inst.K12 == 12'hfff;});   // from randomization
  finish_item(m_utran_inst);

  for (int i=1;i<15;i++)
  begin
    start_item(m_utran_inst);
    assert( m_utran_inst.randomize()
      with {m_utran_inst.opc1   == alur1_op1 && // Specify all the
            m_utran_inst.opc3r   == orr1_op3 && // variables to choose
            m_utran_inst.Rd      == RegD'(i) && // a specific instruction
            m_utran_inst.Uk4     == 4'hf   ;}); //
    finish_item(m_utran_inst);
  end
endtask
```

In this example the *load immediate* instruction **ldir** is used to fill the upper part of register **R1** with the hexadecimal value "fff". Then by iterating over all register addresses, the **alu** instruction **orr1** uses the **R1** value together with "UK4 = f" to fill each slot with the value "ffff". "f" is used due to the nature of the unknown values in simulation forcing the value to "1".

Figure 3.4 and 3.5 shows how the sequences also improve readability for debugging by displaying how and when they are executed as well as providing detailed information about each transaction generated. The readability is further improved with the enumeration of variables, providing quick and easy understanding of the current state. It might not seem all that necessary with only a few sequences,

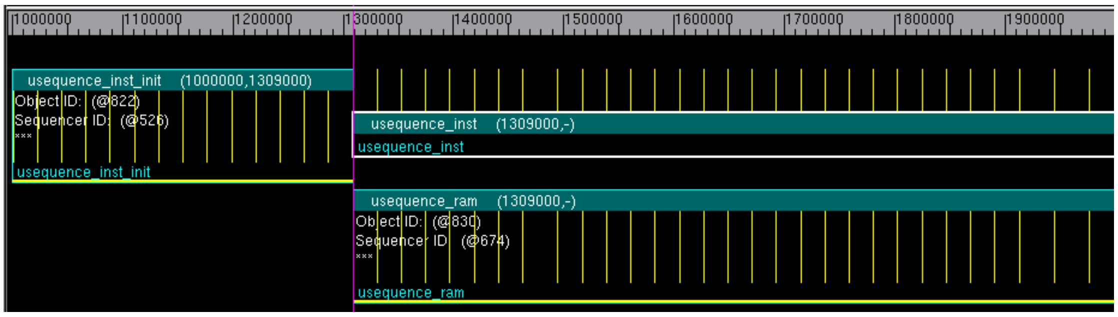
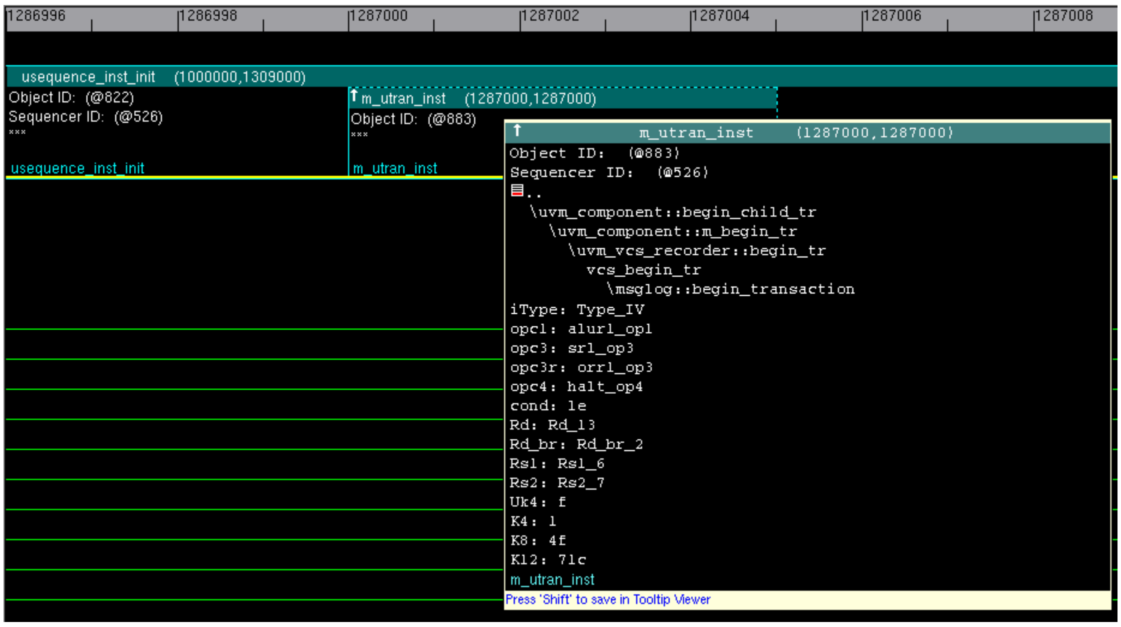


Figure 3.4: Simulation with multiple sequences



```
m_utran_inst    (1287000,1287000)
Object ID:  (@883)
Sequencer ID:  (@526)
..
  \uvm_component::begin_child_tr
    \uvm_component::m_begin_tr
      \uvm_vcs_recorder::begin_tr
        vcs_begin_tr
          \msglog::begin_transaction
iType: Type_IV
opc1: alur1_op1
opc3: srl_op3
opc3r: orrl_op3
opc4: halt_op4
cond: le
Rd: Rd_13
Rd_br: Rd_br_2
Rs1: Rs1_6
Rs2: Rs2_7
Uk4: f
K4: 1
K8: 4f
K12: 71c
m_utran_inst
Press 'Shift' to save in Tooltip Viewer
```

Figure 3.5: Sequences provide detailed transaction information

but imagine having several sequences running at different times in the simulation, then this would surely make it easier to keep track of everything.

For this framework, it is not only important to reduce verification effort, but also promote any feature that can aid in the debugging process as well.

3.3.3 Sequencer

The sequencer implements the handshaking methods used by the sequence and driver, acting like an interagent between production and use of transactions. All the necessary functionality is present in the inherited class requiring no additional implementation, and the sequencer is usually defined with an inline typedef. The only specification is the type of transaction used.

```
typedef uvm_sequencer #(utran_inst) usequencer_inst;
 // Define a sequencer for given transaction
```

3.3.4 Driver and Monitor

Now that we have the input generated it needs to be sent to the DUV for simulation. For the transaction level data to be used as input it needs to be translated down to RTL-level, and it is in the driver and monitor we place the border between TLM and RTL.

The driver is the active part of the verification component and uses the clock input to time the events. It pulls new transactions from the sequencer, then translates this to a bit string, before sending it to the interface. Depending on the type of transaction, it can involve simple single transmissions like in for instruction input, or be more complicated multi-cycle transfers in the case of a bus transmission. In the last case, the driver simply calls a function and leaves the specifics to the interface as described in Section 3.3.5.

```
{}
bit[15:0] inst;
{}
task run_phase(uvm_phase phase);
  forever
  begin
    @(posedge si.clk);
    begin
      seq_item_port.get(m_inst);
      case(m_inst.iType)
        Type_I: inst = {m_inst.opc1,m_inst.Rd,m_inst.Rs1,m_inst.Uk4};
        {...}
      endcase
      si.inst = inst;
    end
  end
endtask: run_phase
```

The code above shows the run phase of the instruction driver. At each new clockcycle a new instruction is pulled from the sequencer and translated to a 16-bit instruction word that is sent to the interface. The case is used to put together the correct variables for the given instruction type.

For the RTL-level values of the DUV to be handled by the UVM environment the previous process must be done in reverse. The monitor reads the values from the DUV through the interface and translates this RTL level data into transactions of the required type. This data can then be broadcast to any subscriber listening to the monitor for processing.

The use of broadcasting and subscribers is another feature that increases structure and reuse of code. By assigning a subscription port, the monitor need not worry about who is listening, but rather broadcast data that can be subscribed to by any other component in the environment.

Since the driver is where RTL-level data gets forwarded to the interface, this is also a good place to add support for direct programming of the CPU. The idea is that the design engineers and the verification engineers could work in the same environment, so by adding support for direct programming of the CPU, it allows design engineers to run simple tests using the same framework. This further increases interoperability and can increase productivity with immediate feedback, as the same monitoring and checking can be utilized.

3.3.5 Interface

The interface is the connection between the UVM environment and the DUV. It provides specific implementations for communication with the current version of the design. For bus transactions, it can implement a bus functional model that executes the proper transmission protocol with the received data. This provides a buffer between the test and the DUV, reducing the requirement of specific knowledge of the current implementation.

3.3.6 Instruction agent

The agent is a container construct used to isolate the set of components referred to as verification components. The sequencer, driver and monitor are the dedicated components dealing with a single type of transaction and interface, and combining these into a single object is another key for reusability. The idea is that they should not need to be aware of anything outside the agent and only deal with the ports provided by the agent. This allows the creation of verification IPs that can be used again in any environment featuring the same protocol.

As the driver and monitor rely on the agent for interaction with the outside, the configuration database is invoked here to provide the interface required.

```
assert(uvm_config_db #(virtual instructionIF)::get(this, "*", "instructionIF", inst_if))
```

In this case the main agent is highly specific and can only be used to drive a CPU using the same instruction set. Other connectors on the CPU, however, have more common data sets like for the RAM interaction.

3.3.7 RAM agent

This is an additional agent that was added in order to have better control of the RAM interactions by the CPU. It is used to monitor the data output and provide randomized or specified input data, enabling proper verification of the store/load instructions. This is only used for four of the instructions, and could be added to the instruction agent, but to avoid mixing code and promote reusability it is better to separate it.

3.3.8 Environment

The environment is the top level UVM class and acts as a container for all UVM components. It is used to instantiate the different components and handles connecting different parts as well as enabling custom configuration provided by the test. It works like a customizable block description, making it easy to build different environments using premade verification IPs. Specific configurations provided by the test can be loaded from the configuration database.

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  m_agent_inst =              uagent_inst::type_id::create ("m_agent_inst", this);
  m_agent_ram =               uagent_ram::type_id::create ("m_agent_ram", this);
endfunction: build_phase
```

The important phase for the environment is the **build_phase** where it instantiates the different components that is used. Other specifications and configurations can also be added here.

3.3.9 Tests

With the environment in place, we can then write tests that use it. Each test is written with a specific goal in mind and can configure the environment to meet certain requirements. The test is where the environment is initialized with the possibility of adding configurations that can modify any part of the system. This is also where you chose which sequences to run, and in what order.

Only simple lines of code are needed to initialize the environment and start the sequences as shown below.

```
{...}
cfg_class.assembly = 0; // Edit custom configuration class
uvm_resource_db#(test_cfg)::write_by_name("test","config",cfg_class,this); // Add
    configuration class to resource database
{...}
phase.raise_objection(this); // Signal start of test

e.initialize(); // Initialize the environment

seqi.start(e.agent_inst.sequencer) // Starts the init sequence on the given sequencer

fork
  seq.start(e.agent_inst.sequencer) // Starts random generation sequence
  seqr.start(e.agent_ram.sequencer) // Starts ram sequence
join

phase.drop_objection(this); // Signal end of test
{...}
```

Objections are used as a test execution mechanism as each class in the environment runs independently. A raised objection signalizes that there is still work to be done, and testing is not finished until all objections have been dropped. The original idea is to put objections in each class to ensure the test is not terminated prematurely, but the recommendation is to reduce the number objections to avoid confusion. Keeping this in the test file only, gives full control to each test.

Due to the level of abstraction each test only needs to specify the environment and which sequences to run, and through the configuration database, it is also possible to specify additional details for specialized tests.

The distinction between the test and the environment that is used is the type of separation that promotes reuse and sharing of work. The engineer writing the test needs no knowledge of how the inner workings of the environment operate or how the sequences are implemented. This allows a higher abstraction-level approach and is unaffected by any modifications done at the lower levels.

3.3.10 Testbench

The testbench is the top layer of a test setup and is where the DUV is connected to the test environment. As described previously the testbench can be used to execute any number of tests that use the same UVM environment.

The configuration database is used here to store references to the required interfaces.

```
uvm_config_db #(virtual instructionIF)::set(null, "*", "instructionIF", inst_if);
uvm_config_db #(virtual ramIF)::set(null, "*", "ramIF", ram_if);
```

3.4 Functional checking in UVM

When it comes to the checking of functionality, this can be implemented in different ways. If we want to keep everything at a higher abstraction level, this is certainly possible. UVM features several classes that aid in the creation of a transaction level verification environment. Especially for designs containing registers there are well defined methods for creating a register model that can mirror the DUV for easy comparison. For a CPU to be properly verified this way it also requires the creation of an Instruction Set Simulator (ISS) to predict the register values and other data like load/store. This was attempted implemented as shown in Figure 3.2 with a scoreboard and checker to compare the predicted values with the actual results.

The advantages of a transaction level checking mechanism is that all of the checking can be done at transaction level, reducing the simulation time when running exhaustive tests. Another advantage is that coverage data can be fed directly to a coverage collector to monitor the progress.

For even more control, the register model can utilize a backdoor connection to the register in the DUV. This allows for direct interaction both for monitoring and even editing of values. However, this takes a toll on the simulation speed as it operates on RTL-level.

The downside of this method is that can be difficult and time consuming to create such models, especially for register models which is often handled by special tools. The creation of an ISS requires an accurate representative model based on the description.

The attempt to implement this checking method was abandoned in favor of ABV, because of complexity and time concerns.

3.5 Functional coverage collection in UVM

To gather data about functional coverage, the coverage collector need only keep track of what stimuli have been placed on the input of the design. This is easily obtained by subscribing to the broadcast port of the monitor to get each transaction that is passed.

```
covergroup inst_cg_type_I; // Create covergroup
  option.at_least = 1; // Specify coverage goal
    op_opc1: coverpoint opc1 // Create coverpoint
    {
      bins ori4 = { ori4_op1 }; // Create bins for
      bins add2 = { add2_op1 }; // the relevant
      bins sub2 = { sub2_op1 }; // variables
    }
    cp_rd: coverpoint Rd;
    cp_rs1: coverpoint Rs1;
    cp_uk4: coverpoint Uk4;
    cross cp_opc1, cp_rd;  // Check cross coverage
    cross cp_opc1, cp_rs1; // for all possible
    cross cp_rd, cp_rs1;   // combinations
endgroup
```

An example of a covergroup with its coverpoints for one type of instructions is shown in the code above. The covergroup is sampled each time its instruction is detected by the monitor, gathering data on value ranges and combinations. This covergroup will be able to show, if the goal is reached, that each instruction of type_I have been tested with every possible combination of variables at least once. It becomes near impossible to complete this kind of cross coverage when the size of variables increase, but in this case the ranges are limited making a complete coverage possible as is shown in Chapter 6.3. For larger cases it is more important to make sure corner cases are reached and that a significant selection of combinations is visited.

The ability to monitor coverage data during runtime also enables this to be fed back into the input generator. Generation can then be actively adjusted during the simulation to increase coverage in important areas while avoiding overexposure of less significant parts. This was not prioritized for the framework as it was not hard to get coverage of the small designs, but it is a necessary feature that should be added.

3.6 Reusability

The advantage of writing reusable code is well known, especially with object oriented languages, but it may be even more essential to think reusability when working with verification.

As UVM was developed with reuse in mind it is natural to assume that all of the components are highly reusable. But considering the target for this framework, let's examine exactly how these components can be used again for different CPU designs.

For input generation the instruction set is the main variable, and we would need an easy way to swap this out. Functional checking depends on how the TLM models are implemented, and these are highly dependent on the design. Functional coverage collection is also heavily design dependent, but the component implementations can be more generic.

If we look at the core component, the transaction object, which is where we specify the variables to be randomized as well as their constraints, it would be possible to make a generic object featuring a generic instruction set structure and import a configuration class containing the instructions and constraints. But this seems rather pointless as the transaction already is an inherited class and contains no other information than what is specific to the current design. Instead the transaction object should be the interchangeable part and contain all information relevant to the specific instruction set.

Both the sequence and the sequencer are specified by the transaction that they use, but other than that contain no other design specific information. The generic implementations need only minor modification to be reused, which could possibly be made a configuration option. Additional custom sequences can easily be added if needed.

The driver is also fairly generic, but depending on how the translations are handled, it requires som editing. The main mechanics of the driver can be used for any design, and the support for assembly input is also design independent.

The monitor would also be rather generic depending on where the translation for transactions is implemented. Its only function is to read the instruction from the interface, then unpack it into a transaction object, and then forward the data to the ISS, register model and coverage collector. The change required lies in the type of transaction used, and how this is decoded, while the the mechanics stay the same.

The ISS may have some reusability in the way it interacts with the other components, but is essentially a unique model for the given design. For the register model, reusability depends on the tools used to create it, and is for this reason highly reusable with the proper tool in place.

The coverage collector is partly reusable in its communication, but requires rewriting of coverage classes relevant to the instruction set.

Chapter 4

Alternatives and improvements

A pure UVM implementation seems like a natural choice for the framework considering the many advantages in terms of reusability. However, some of the parts require significant effort to implement may not be sufficiently reusable. Here we take a look at an alternative, which is to move the functional checking outside the environment and instead handle this with assertions.

4.1 ABV vs UVM checkers

Instead of the TLM checking described in the last chapter it is possible to use a separate set of assertions for this purpose. Using the *bind* functionality from SystemVerilog to connect to the design at RTL-level enables direct monitoring of any signals, and assertions can be written to trigger on any value change.

It might seem like we take a step back when we don't use a higher abstraction level for the checking as well, but instead resort to directly targeting RTL level behavior. One argument for this step back is the control you gain when writing RTL level assertions. When you write a model that emulates the design you might be left with the same problem as before; how do you know that your model is correct? You would need some way to verify this. The assertions allow this to be done with a separate set of eyes. You look at the problem from a slightly different angle and target specific areas directly. This makes it easier to confirm that the checking is done correctly.

One of the advantages gained from using ABV is the direct feedback during simulation making it easy to instantly see how well the design works. Figure 4.1 shows an example of the assertion response during simulation where each upward green

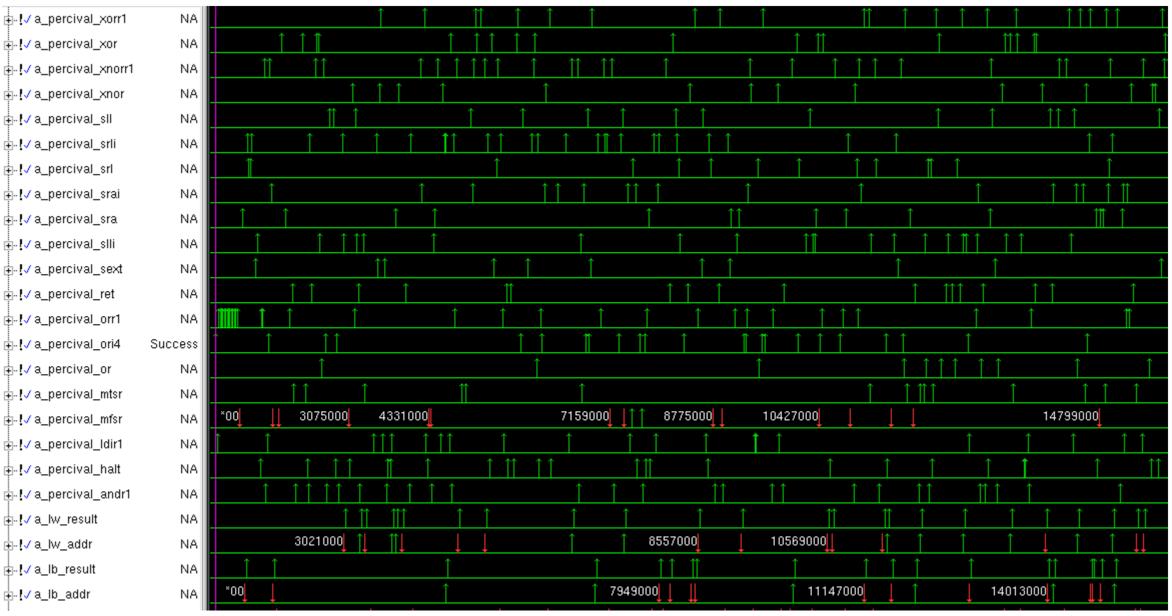
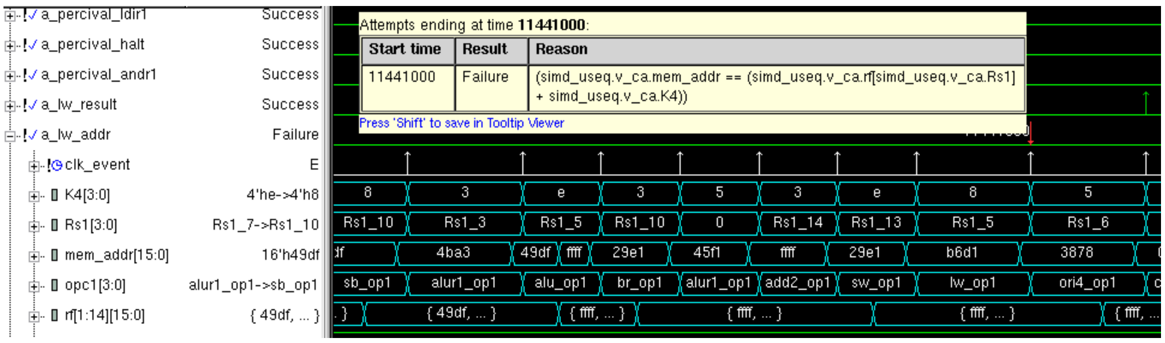


FIGURE 4.1: Example of assertions

arrow indicate an assertion that passed. This effectively makes the debugging process easier as you can more easily pinpoint any errors. Figure 4.2 show how the failed assertions in red give detailed information on where the error is and what went wrong.


Attempts ending at time **11441000**:

Start time	Result	Reason
11441000	Failure	(simd_useq.v_ca.mem_addr == (simd_useq.v_ca.rf[simd_useq.v_ca.Rs1] + simd_useq.v_ca.K4))

Press 'Shift' to save in Tooltip Viewer

a_percival_ldir1 Success
a_percival_halt Success
a_percival_andr1 Success
a_lw_result Success
a_lw_addr Failure
clk_event E
K4[3:0] 4'he->4'h8
Rs1[3:0] Rs1_7->Rs1_10
mem_addr[15:0] 16'h49df
opc1[3:0] alur1_op1->sb_op1
rf[1:14][15:0] { 49df, ... }
8 3 e 3 5 3 e 8 5
Rs1_10 Rs1_3 Rs1_5 Rs1_10 0 Rs1_14 Rs1_13 Rs1_5 Rs1_6
4ba3 49df ffff 29e1 45f1 ffff 29e1 b6d1 3878
sb_op1 alur1_op1 alu_op1 br_op1 alur1_op1 add2_op1 sw_op1 lw_op1 ori4_op1
{ 49df, ... } { ffff, ... } { ffff, ... } { ffff, ... } { ffff, ...

FIGURE 4.2: Example of failed assertion

Another advantage is the overview gained during the verification process. Each assertion deals with one specific event and can be treated independently. This enables work to be more easily shared between verification engineers.

A significant downside is the decrease in simulation speeds. Doing all the checking at RTL-level takes a toll on the simulation, as well as the large amount of assertions that has to be evaluated actively. Because of this it is important to consider what method is best depending on the size and type of design that is being verified. For small designs like the ones used with this framework, the use of ABV gives an advantage for overview and control of the checking, and the simulation time is not as relevant. But for larger scale and different design types it might be more productive to keep the abstraction level raised.

There is however potential to improve the ABV, not necessarily for simulation speed, but reduced effort when writing assertions. The next section will look at possible improvements in terms of reduced code size and increased reusability.

4.2 ABV reusability

Writing assertions tend to generate a significant amount of repeated code as the properties do a lot of the same work, with some variations. Though assertions seems to be highly independent there is also potential for enabling a fair amount of reusability as well. There are two types of reusability considered here: internal reuse, reducing repeating of design specific code, and external reuse, parts of the framework that can be used for different designs.

One method for internal reuse is the use of an assertion interface in the same way as the interface for the inputs, providing a relative reference to the signals that are monitored. This is useful when the design is under development as signals and registers may be subject to change. As there are several assertions monitoring the signals, the lack of a relative reference would lead to unnecessary editing in several files. This interface is implemented in the block labeled 'common assertions' in Figure 4.3.

```
Property p_add_result();
  (v_ca.opc1==add_op1, // Detect 'add' opcode
    Rs1 = v_ca.Rs1;   //
    Rd = v_ca.Rd;     // Store input values
    Uk4 = v_ca.Uk4;)  //
    ##1 // Wait one clock cycle
    (1,rf = v_ca.rf;) // Store current register values
      |-> ##1       // Wait for results to be ready
      v_ca.rf[Rd] == rf[Rd] + (rf[Rs1] | Uk4); // Check result
endproperty
```

If we take a look at the assertion properties for the generic arithmetic operation add, we notice that it consist of two parts. One part is the functional check, where we have two values that should be added. The other part is the temporal information. This covers when the values should be read, and when the result is ready for comparison. It would seem it should be possible to separate these two properties, and since many operations like arithmetic operations share the same sequential timing, this could be extracted for reuse. This leads to three potential improvements;

First, when dealing with processor designs there are several basic operations that are essential to have. Additionally there are many other common operations as well. Instead of having to specify the details of these operations each time you write such assertions it would be easier to create a package containing predefined functionality for common operations.

```
function automatic regsize alu_and (regsize a, b, hbyte_p c);
  alu_and = a & (b | c);
endfunction

function automatic regsize add (regsize a, b, hbyte_p c, bit carry);
  add = a + (b | c) + carry;
endfunction
```

The functions shown above are examples of generic operations that you would find in most processor designs. Generic here means that these can be configured to any parameter sizes as well as having optional inputs. A complete set of such functions could reduce effort when writing assertions, and also enable automatic generation of assertions, which is described later. This does not only provide internal reuse, but as these are common functions this can be used for other designs as well.

Secondly, we have sequences, not to be confused with the UVM sequences. This is a much used feature of ABV that allow complex sequential statements to be compacted into sequence variables. An example of this is shown below. This is a constructed example, as no complex timings are required for the implemented instructions, but it also shows use of the package function shown above. The sequence 's1' can represent any normal or contitional delay, and may be subject to change during design development.

```
sequence s1;
  first_match(A ##[1:3] B);
endsequence

property add;
  (v_ca.opc1==add_op1) // Detect 'add' opcode
    {...} // Store input values
    s1    // Wait for s1
    {...} // Store current register values
    |-> s2 // Some other possible delay
    v_ca.rf[Rd] == add(rf[Rd],rf[Rs1],Uk4,0); // Use package function to check result
```

Though the use of sequences is a common practice, in terms of reusability it is another method for internal reuse and can be utilized to great benefit. When you have a sequence like this repeated in several assertions and for some reason the timing of this needs to be changed, then you would have to modify every instance. For this reason it makes sense that any sequence used more than once, can and should be extracted into a package to allow ease of editing as well as reduction of code size.

This may not have a significant impact for the targeted processor designs, as there are no complicated timings to take care of, but bigger systems with more intricate sequential conditions can have a larger benefit from this.

Thirdly, we have the possibility of creating a script to generate the assertions. The functional part of each assertion describes an expected result in either a register or output after a certain event has been detected. It may be a result of an arithmetic operation, a comparison when moving or loading data, or just the toggling of a signal like a chip enable. If we don't consider the timing of these events, it should be possible to generate these functional assertions based on a functional description. Having a proper library of CPU functions could allow new assertions to be generated quickly for new designs.

This was considered to be created along with the framework, but lack of time prevented this.

4.3 Functional Coverage collection

Functional Coverage collection is made even easier when using assertions instead of higher level models. The reason is that instead of having to create a system for detecting and triggering the coverpoints, this has already been taken care of by the assertion. The coverage can be collected each time an assertion is triggered with very little added code.

Another advantage here is that it is possible to collect coverage information of more that what is available on I/O interfaces. Since the assertions monitor the design directly on RTL-level, coverage can also be checked for internal signals.

4.4 Complete framework

Based on this and the previous chapter, the final framework has been built using UVM for input generation, timing and synchronization, and added functional checking and coverage as a separate part. The resulting structure is illustrated in Figure 3.2.

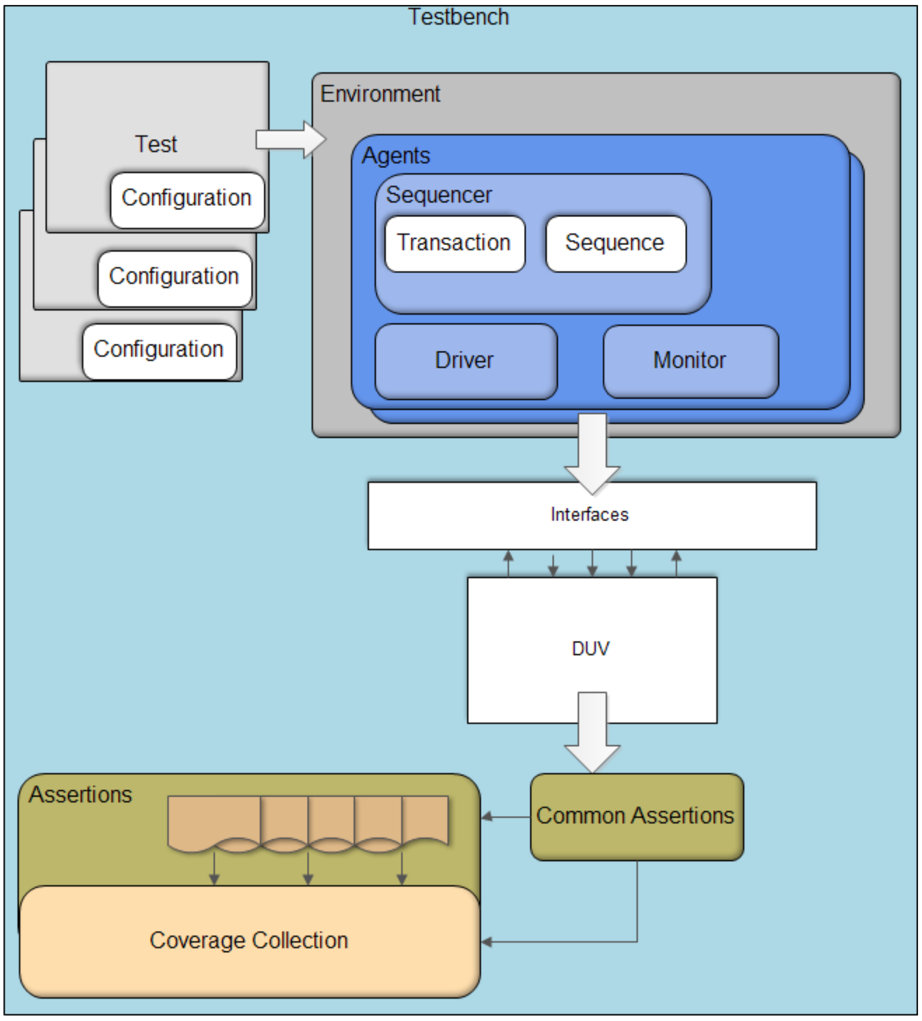


FIGURE 4.3: Complete framework

Chapter 5

Reusability: Adapting to new designs

Now that we have a complete functional framework for the synchronous processor design we can look at how this can be reused for different processor designs. From the discussion on reusability in the Chapter 3.6 it is clear that most of the UVM environment is fairly generic and can be reused with only minor modifications. The assertions, being inherently design specific, will naturally need to be rewritten, but the effort can be reduced by the use of standardized packages. The coverage collection, being strongly linked to the assertions, will also need rewriting, but following a simple template effort can be reduced. First the framework will be adapted to work with the asynchronous design, and then the stack processor will be considered.

5.1 Asynchronous processor

The goal of [17] was to convert the synchronous processor used previously to asynchronous logic. The intention is to have a design that performs the same operations as the synchronous version, meaning that seen from the I/O perspective it is identical. Because the asynchronous design uses the same instruction set and is essentially the same seen externally, the only modification required is in components that are affected by the clock. Removal of the clock warrants a new trigger for timing of new instructions as well as knowing when the results are ready in the registers.

It is easy to think that removing the clock will complicate the verification process because of the lack of synchronization making it hard to time events. However, the instruction generation in the UVM environment operates on a request basis, similar to asynchronous, meaning that the input can easily be generated asynchronously. The only place the clock is used in the environment is in the driver, where the requesting is done. Replacing the clock with the signal that the processor uses to request new instructions allows most of the system to remain the same.

There are minor adjustments that must be made to accompany the added I/O signals, but these can be easily handled by the instruction and ram drivers. These involve request/grant communication with the memories for instructions and data and can simply be responded to with optional delays.

The previous consideration also holds true for the instruction assertions checking register values. Evaluating the assertion at the request of a new instruction is the same as using a clock, and the design ensures that results are ready by the time it requests a new instruction. Figure 5.1 show how the signal timing is unaffected by the change, and that triggering the next instruction on the *ram_clk* produced by the DUV is essentially the same as using a clock with varying clock cycle. Assertions written for other signals and events may require some extra attention, but for the most part there are internal request/acknowledge signals that can be used as triggers replacing the clock.

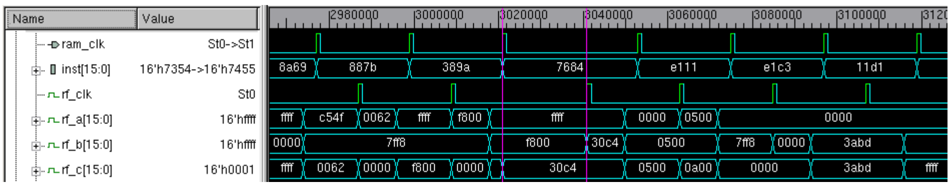


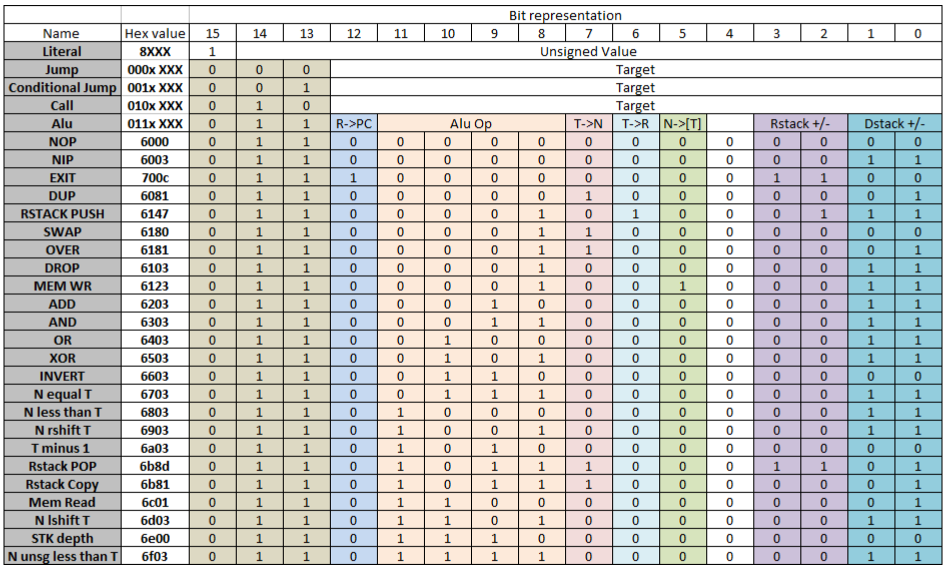
FIGURE 5.1: Timing

The assertions used for the synchronous version could be completely reused as these were mostly register related. Adding extra assertions to fully verify the asynchronous version was not prioritized as the focus was aimed more at proof of concept. The same considerations hold true for coverage that can also be reused here.

5.2 Stack processor

Featuring a completely different instruction set as well as a conceptually different internal operation, the stack processor challenges the reusability of the framework. A new instruction set means changes to the way instructions are generated, thus the transaction object needs to be rewritten. This again leads to the rewriting of assertions as the instructions operate differently, and also to monitor coverage we need to incorporate the new instructions.

Due to the way the new instruction set is built it is not as easy to extract variables for randomization as it was for the previous set. As shown in Figure 5.2 the instructions are not as easily enumerated, but the limited number of instructions currently implemented, allow these to be easily specified individually. It is still possible to isolate instruction types from data and randomize the inputs properly.



		Bit representation															
Name	Hex value	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Literal	8XXX	1	Unsigned Value														
Jump	000x XXX	0	0	0	Target												
Conditional Jump	001x XXX	0	0	1	Target												
Call	010x XXX	0	1	0	Target												
Alu	011x XXX	0	1	1	R->PC	Alu Op				T->N	T->R	N->[T]		Rstack +/-		Dstack +/-	
NOP	6000	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
NIP	6003	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1
EXIT	700c	0	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0
DUP	6081	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1
RSTACK PUSH	6147	0	1	1	0	0	0	0	1	0	1	0	0	0	1	1	1
SWAP	6180	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0
OVER	6181	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	1
DROP	6103	0	1	1	0	0	0	0	1	0	0	0	0	0	0	1	1
MEM WR	6123	0	1	1	0	0	0	0	1	0	0	1	0	0	0	1	1
ADD	6203	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	1
AND	6303	0	1	1	0	0	0	1	1	0	0	0	0	0	0	1	1
OR	6403	0	1	1	0	0	1	0	0	0	0	0	0	0	0	1	1
XOR	6503	0	1	1	0	0	1	0	1	0	0	0	0	0	0	1	1
INVERT	6603	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0
N equal T	6703	0	1	1	0	0	1	1	1	0	0	0	0	0	0	1	1
N less than T	6803	0	1	1	0	1	0	0	0	0	0	0	0	0	0	1	1
N rshift T	6903	0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	1
T minus 1	6a03	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0
Rstack POP	6b8d	0	1	1	0	1	0	1	1	1	0	0	0	1	1	0	1
Rstack Copy	6b81	0	1	1	0	1	0	1	1	1	0	0	0	0	0	0	1
Mem Read	6c01	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	1
N lshift T	6d03	0	1	1	0	1	1	0	1	0	0	0	0	0	0	1	1
STK depth	6e00	0	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0
N unsg less than T	6f03	0	1	1	0	1	1	1	1	0	0	0	0	0	0	1	1

FIGURE 5.2: Instruction set for the stack processor

Other than swapping out the transaction and specifying this in the sequence and sequencer, no further changes are required for the input generation unless an init sequence is needed.

Functional checking and coverage require a bit more work. None of the previous assertions or covergroups can be used for this. But if we make use of the reusable package for assertions, the amount of work can be reduced. Together with a template that includes coverage for the given assertion, only the temporal information is required for the most common instructions.

There were not many assertions added for this design as the focus was primarily to enable input generation and show the potential for future verification.

Chapter 6

Evaluation

This chapter will evaluate the resulting framework, summing up the advantages gained from using its use and look at how it performs.

In the previous chapter, the framework was shown to be highly reusable and able to handle both synchronous and asynchronous logic. To evaluate the potential of the framework we have to look at its reusability and ability to increase productivity. We should also look at the metrics that can be extracted from the simulations, which typically include coverage count, and the increased amount of bugs uncovered.

6.1 Reusability

As shown in the previous chapter, large amounts of the framework can effectively be used for multiple designs. The highly standardized structure of the UVM is the primary reason for a high degree of reuse for input generation. It can be argued that creating random instructions is a rather simple task, but from what has been shown, any type of complex sequence can easily be created. The other parts of the verification process, however, are more specialized towards the DUV and cannot just be duplicated. But as discussed in Chapter 4.2 it is possible to extract similarities that can be used again.

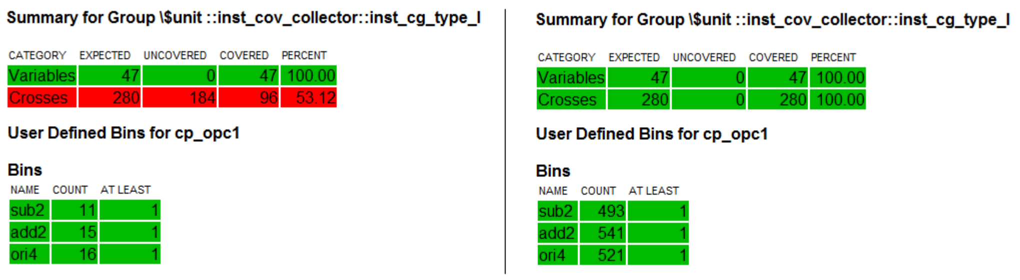
6.2 Productivity

Reusability is one of the keys for increasing productivity as it reduces the amount of redundant work. The structure of the UVM again removes much of the tedious housekeeping, and with a framework implemented, focus can be kept on the important tasks. The constraint random generation further promotes this by simplifying the generation process, allowing the main attention to be directed at checking for errors. The productivity for assertion writing will also be increased if most of the standard assertions could be generated automatically.

6.3 Coverage

The coverage metric shows how thoroughly a design has been verified and is divided into two parts: The code coverage, showing that all lines of the design has been run at least once indicating whether all states are reachable and if the test was thorough enough to reach all states. The functional coverage shows how many times each assertion is triggered and how the covergroups are sampled.

As assertions and covergroups are only added to show the concept, they are not complete enough to show the functional coverage of the designs. But an example can be used to help illustrate the potential reach of contstraint-random verification. Figure 6.1 show two simulations where the coverage for instruction type_I of the synchronous CPU was measured. The first test was run for a short time sdf all instructions several times, but not that many combinations of input values was tested. The second simulation was allowed to run until all the specified combinations was reached, showing that the only limiting factor for a more complete verification is time.



Summary for Group \$unit ::inst_cov_collector::inst_cg_type_I

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	47	0	47	100.00
Crosses	280	184	96	53.12

User Defined Bins for cp_opc1

Bins

NAME	COUNT	AT LEAST
sub2	11	1
add2	15	1
ori4	16	1

Summary for Group \$unit ::inst_cov_collector::inst_cg_type_I

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	47	0	47	100.00
Crosses	280	0	280	100.00

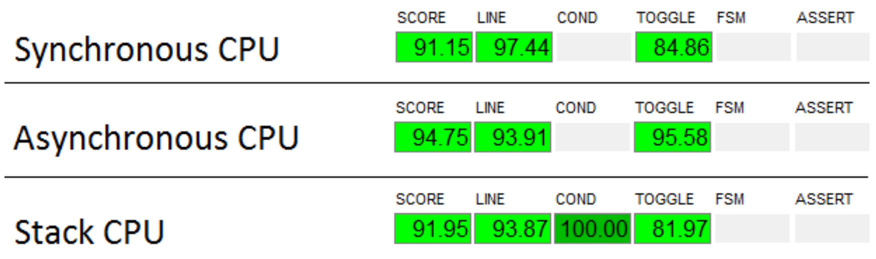
User Defined Bins for cp_opc1

Bins

NAME	COUNT	AT LEAST
sub2	493	1
add2	541	1
ori4	521	1

Figure 6.1: Coverage results for one covergroup

The code coverage is a more descriptive result as it shows if we are able to generate inputs that exite all of the design. Figure 6.2 shows that all the designs were stimulated to reach over 90% of the code. The remaining percentage is mostly due to several default states that are never executed when all states are implemented.



	SCORE	LINE	COND	TOGGLE	FSM	ASSERT
Synchronous CPU	91.15	97.44		84.86		
Asynchronous CPU	94.75	93.91		95.58		
Stack CPU	91.95	93.87	100.00	81.97		

FIGURE 6.2: Coverage

6.4 Bugs

Uncovering bugs or proving the lack of these is the naturally main goal of verification. However, as the checking is not fully implemented, the amount of bugs uncovered is of no real value. But some of the special cases are worth mentioning.

One case where the real value of constraint random testing became obvious was one of the instructions that would produce correct results most of the time, but sometimes it would fail. Figure 6.3 show how the assertions trigger pass and fail for this specific instruction.

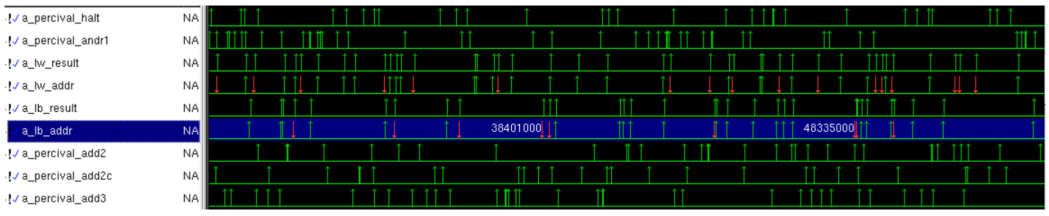


FIGURE 6.3: Example of bug

The reason for this bug was certain residue input on the alu from previous instructions that would be added to the calculation. A bug like this would not be easy to spot without randomized testing.

Chapter 7

Conclusion and future work

A highly reusable verification framework has been made in order to verify three different processor designs. The framework was shown to be relatively easily adapted to the different designs with minor adjustments. It was created using a combination of the Universal Verification Methodology and System Verilog Assertions. UVM was shown to have significant advantages for handling input generation at the transaction level with methods in place for timing and synchronization. The use of UVM for functional checking, however, was considered to be less efficient in terms of interoperability and reusability. For functional checking ABV was considered to provide a better overview as well as ease work division.

The framework was built based on a simple synchronous processor design. An asynchronous version was used to see how the lack of clock would affect the verification process. This did not have a huge impact on input generation due to the way the UVM communicates. The assertions and coverage needed new triggers for timing events, but being directly connected to the DUV, this was easily extracted. Lastly a stack processor was targeted featuring a different instruction set. Again, the input generation was rather easily implemented after adding the new instructions. Assertions and coverage was only briefly explored as significant knowledge is required to verify fully.

The framework was able to stimulate each of the designs with over 90% code coverage, with the remainder being mainly default states. The functional coverage was inconclusive as not a complete set of assertions was implemented, but rather a subset included to show a proof of concept.

UVM has been shown to be an efficient methodology for verification, but there are some inefficient macros that should be avoided for best performance.

To increase the potential for reuse of the assertions, the temporal and functional parts were separated and extracted into a package that can be used again to write or generate assertions for common CPU functions.

7.1 Future work

- The implementation of assertions and coverage need to be expanded for the designs to be verified properly.
- Other UVM classes can be explored to increase the frameworks potential.
- Specifically the addition of virtual sequences for increased simulation control.
- Complete implementation of both TLM and ABV checking for simulation speed evaluation.
- Create script for automatic generation of assertions.
- Use the framework for a larger scale design to see the full potential.

Bibliography

[1] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–560, 2006.

[2] IEEE Standard for the Functional Verification Language E. *IEEE STD 1647-2008*, pages c1–464, Aug 2008.

[3] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[4] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013.

[5] Harry Foster. "Wilson Research Group Functional Verification Study 2012". *http://blogs.mentor.com/verificationhorizons/blog/author/hfoster.*

[6] Ronan Barsic. *CTM16 instruction set.* Atmel, 2013.

[7] Milos Becvar and Greg Tumbush. Design and Verification of an Image Processing CPU using UVM. In *DVCon*, 2013.

[8] Jonathan Bromley. If SystemVerilog is so good, why do we need the UVM? Sharing responsibilities between libraries and the core language. In *Specification Design Languages (FDL), 2013 Forum on*, pages 1–7, Sept 2013.

[9] W. Ecker, V. Esen, L. Schonberg, T. Steininger, M. Velten, and M. Hull. Impact of Description Language, Abstraction Layer, and Value Representation on Simulation Performance. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[10] Adam Erickson. Are ovm & uvm macros evil? a cost-benefit analysis. In *Proceeding of Design and Verification Conference (DVCON)*, March 2011.

[11] Peter Flake. Why SystemVerilog? In *Specification Design Languages (FDL), 2013 Forum on*, pages 1–6, Sept 2013.

[12] Accellera Systems Initiative. ”UVM (Universal Verification Methodology)”, [Online]. Available: http://www.accellera.org/downloads/ standards/uvm.

[13] Yangyang Li, Wuchen Wu, Ligang Hou, and Hao Cheng. A Study on the Assertion-Based Verification of Digital IC. In *Information and Computing Science, 2009. ICIC ’09. Second International Conference on*, volume 2, pages 25–28, May 2009.

[14] N. Sudhish, B.R. Raghavendra, and H. Yagain. An Efficient Method for Using Transaction Level Assertions in a Class Based Verification Environment. In *Electronic System Design (ISED), 2011 International Symposium on*, pages 72–76, Dec 2011.

[15] Synopsys, Inc. *OpenVera^{TM}*. http://www.open-vera.com.

[16] Allan Green Vargas. *Ultra-low Power Stack Based Processor for Energy Harvesting Systems.* (unpublished master’s thesis) NTNU, 2014.

[17] Bjørn Thomas Søreng Vee. *Conversion of a Simple Processor to Asynchronous Logic.* (unpublished master’s thesis) NTNU, 2014.