Simulation-Based Verification of System Level Models

Verification of a RISC-V processor in a UVM environment

Nawras Altaleb

Department of Electrical and Computer Engineering
Technische Universität Kaiserslautern

This dissertation is submitted for the degree of
Master

April 2019

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Nawras Altaleb
April 2019

Abstract

In This work we introduce a new level of abstraction to the PDD methodology splitting system-level modeling into abstract and complex models. The abstract model will represent the desired behaviour of the system while the complex model will project the designer's choice of implementing this behaviour. The reason for this, is to save the effort of refining the properties of the abstract model, generated by PDD, when verifying RTL level, which is a time consuming process and prone to errors. We also introduce a coverage driven verification methodology to verify the correct behaviour of the complex system-level model compared to the abstract one, referred to as Golden Model. This simulation-based methodology uses UVM-SystemC and indicates how much code coverage is achieved with running simulations.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

AM	Abstract Model
AST	Abstract Syntax Tree
BB	Basic Block
CDFG	Control-Data Flow Graph
CDV	Coverage Driven Verification
CLINT	Core Local INTerruptor
CSR	Control and Status Register
ESL	Electronic System Level
ESL	Electronic System Level
GM	System Level Golden Model
MEIP	Machine External Interrupt
MSIP	Machine Software Interrupt
MTIP	Machine Timer Interrupt
PDD	Property-Driven Design
PE	Process Element
PLIC	Platform-Level Interrupt Controller
PPA	Path Predicate Abstraction

SCAM	SystemC Abstract Model
SCV	SystemC Verification library
SER	Specify, Explore-and-Refine
SL	System Level
UVC	Universal Verification Component
UVM	Universal Verification Methodology

Chapter 1

Introduction

Embedded-systems design process have been facing a lot of challenges, due to the rapid growth in systems complexities, which renders the traditional design methods infeasible.This leads to the well-known productivity gap generated by disparity between the rapid paces at which design complexity has increased in comparison to that of design productivity [7]. One of the commonly-accepted solutions for closing this productivity gap is to raise the level of abstraction in the design process and apply design-automation techniques for modeling, simulation, synthesis, and verification to the system design process.

On the modeling and simulation side, several approaches exist for the virtual prototyping of complete systems which can be assembled at various levels of detail and abstraction. Such simulation-centric approaches enable the horizontal integration of various components in different application domains. However, approaches for the vertical integration for system synthesis and verification are limited (e.g., SCAM) and there are no commercial solutions available yet.

In the following section we will explain the different abstraction levels involved in system design.

Abstraction Levels

In order to bridge the productivity gap in system design process, design methodologies and tools were forced to move to higher level of abstraction. We will briefly explain the relationship between different design methodologies on different abstraction levels [7].

Processor-Level Behavioral Model

On the processor level, we define and design computational components or *processing elements* (PEs). Each PE can be a dedicated or custom component that computes some specific function. The functionality or behavior of each PE can be specified in several different ways, most common of all uses *Control-Data Flow Graph* (CDFG), which can represent any programming-language code. CDFG consist of *if* diamonds, which represent *if* conditions and Basic Blocks (BBs), which represent computation.

System-Level Behavioral Model

Processor-level behavioral models such as CDFG can be used for specifying a single processor, but will not suffice for describing a complete system that consist of many communicating processors. A system-level model must represent multiple processes running concurrently in SW and HW. In order to represent this concurrency, we need a synchronization mechanism for data exchange, such as the concept of a channel, to encapsulate data communication. Also, we need a model which support hierarchy, as so to allow designers to write complex system specifications without difficulty.

In chapter 2 we will present system-level abstraction methodology and the division of this level into two sub-levels; abstract system-level representing the general behaviour of the design and complex system-level representing the detailed design at the *Electronic System Level* (ESL) of the design following designers implementation choices. While the abstract system-level can be built using a single processor (i.e. single CDFG), the complex model will most likely include more than one concurrent processor.

System Design Methodology

Design flow has been changing with the increase in system complexity over the past half-century. [7] sums this change within three periods, as shown in Figure 1.1.

Through the first two periods, *Capture-and-Simulate* and *Describe-and-Synthesize*, designers were still facing the system gap between SW and HW, and therefore between specification and implementation. In such design flows there were too many levels of abstraction between system specification and gate level design for SW designers to get involved.

Specify, Explore-and-Refine (SER), on the other hand, introduce a methodology that include both SW and HW. It starts with executable specifications, on the system-level (SL), that represent the system behavior and extends it to include several models with different details. Each model is used to prove some system property and is considered as a specification for next level model, in which more implementation detail is added. This methodology can be also described as a *top-down Methodology*.

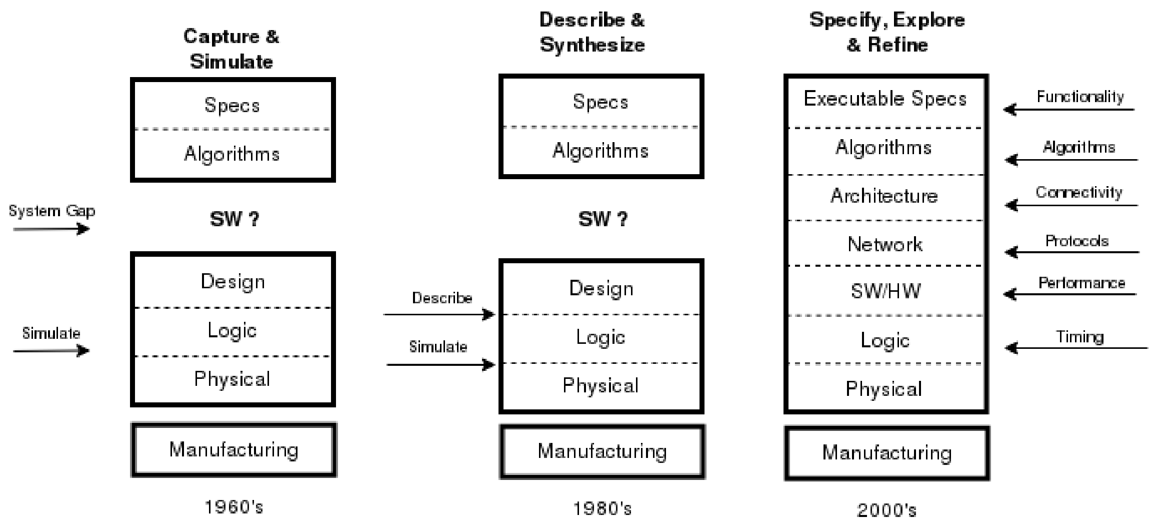


Fig. 1.1 Evolution of design flow.

Chapter 2 will include a description of the methodology and semantics followed in this work and the various modeling levels used in it, including our added system-level model. Chapter 3 will introduce the simulation-based verification methodology applied on this added system-level, while chapter 4 will discuss the coverage metrics needed to indicate a successful verification. And finally, chapter 5 will conclude our work in a summary and present some of the open topics still in question.

Chapter 2

Property-Driven Design Methodology

Every design methodology contains models at various steps of the flow that represents a certain aspects of reality while hiding others that are irrelevant or not yet known. In each design step, a refinement tool takes the input model and implements a set of design decisions in order to generate an output model at the next lower level of abstraction. In general, both refinement and decision-making can be manual or automated. However, tools for automation of these design processes beyond simulation can only be applied if models and corresponding abstraction levels are well-defined with clear and unambiguous semantics.

In this chapter we will talk about the Property-Driven Design (PDD) methodology used in our work and the SystemC-PPA semantics used with it. We will also introduce one level of abstraction along this methodology with a few examples built using this methodology.

2.1 Property-Driven Design

PDD is a design methodology that bridges the semantic gap between abstract SW model and HW model by establishing a well-defined and sound relationship between an abstract system-level model written in SystemC and the RTL [20]. Basically, PDD ensures that the system-level model, at all times, remains a so-called Path Predicate Abstraction (PPA) of the RTL implementation, We call this "PPA-designable subset" of the language *SystemC-PPA* semantics.

PDD treats system-level model as golden design model instead of just a prototype. allowing it to move the complex verification tasks from the RTL to the system level where verification of global behaviors is usually much easier and less costly. This process happens by automatically generating a set of abstract properties from the system-level model and then applying refinements to this set of properties along the creation of the RTL implementation. As a result, this methodology ensures that the RTL is a correct *refinement* of the ESL, and the

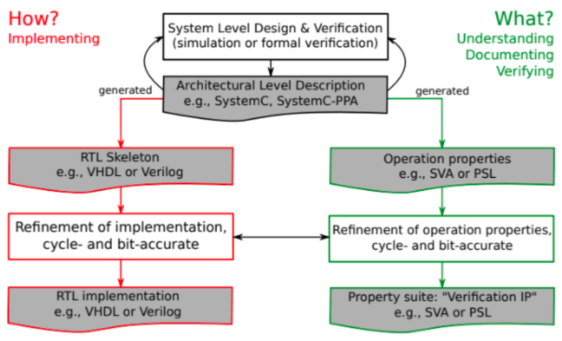

How?
Implementing
What?
Understanding
Documenting
Verifying
System Level Design & Verification (simulation or formal verification)
Architectural Level Description e.g., SystemC, SystemC-PPA
generated
generated
RTL Skeleton e.g., VHDL or Verilog
Operation properties e.g., SVA or PSL
Refinement of implementation, cycle- and bit-accurate
Refinement of operation properties, cycle- and bit-accurate
RTL implementation e.g., VHDL or Verilog
Property suite: "Verification IP" e.g., SVA or PSL

Fig. 2.1 Property-Driven Design flow

ESL is a *sound* abstraction of the RTL. The term *soundness* describes a well defined formal relationship between an ESL and an RTL model.

Figure 2.1 shows the general flow of property-driven top-down design. It starts with a specification at the *Electronic System Level* (ESL). formalized using SystemC-PPA semantics. The RTL design process splits up into two concurrent tracks. The RTL implementation track (left side of Fig. 2.1) is concerned with how the specified functions are implemented. The verification track (right side of Fig. 2.1) maintains the correctness of the implementation by verifying the property suite against the RTL model: it checks what behavior has been implemented.

The starting points for both tracks, the RTL design and the property suite, are generated automatically from the SystemC-PPA ESL description with the provided open-source software tool SCAM [13]. This has the advantage that global design decisions can be made already at the electronic system level. If the ESL design works correctly so will the RTL design implemented from it. However, also any system-level bug will appear in the RTL, as ESL and RTL are no longer decoupled. Thus, it's important that the ESL is thoroughly verified before PDD-based RTL implementation is started.

2.2 ESL Modeling With SystemC-PPA

The system-level model is executable, i.e., it can be simulated with any SystemC simulator. Communication between the modules in a system is modeled on the transaction level, i.e., the system behavior is given by time-abstract finite state machines described at word level. The FSMs send each other messages based on synchronization events. Each FSM is a PPA and describes one module. The FSM states represent communication events, and the transitions between states represent operations. The overall system behavior results from an asynchronous product of the individual FSMs. This allows for a modeling of all interleavings of messages being passed between the modules, and ensures capturing all possible behaviors of the system. Due to the untimed behavior of the system-level model, each module is allowed to run at its own speed. In order to exchange a message between two modules, they need to synchronize through a handshake.

At system level this handshake is implemented through events. During the RTL design process, the handshake is realized by a four-phase handshake. Sometimes, implementing full four-phase handshaking bears unnecessary overhead, e.g., in cases where loosing a message is acceptable. SystemC-PPA, therefore, provides three different kinds of communication interfaces called *ports*. The three supported interfaces provide enough flexibility to handle any hardware communication needs. The type of communication interface is selected during the system-level design process.

Each interface generates a different kind of property suite and, therefore, affects the hardware design process. The basic interface is called *blocking* and implements a blocking message passing handshake. It ensures that a message is never lost. *MasterSlave* is a special case of the blocking interface for synchronous communication. If it is known that one side (the *slave*) is always ready for communication then the other side (*master*) may communicate without waiting for synchronization. Finally, the *Shared* interface models the behavior of a volatile memory.

In order to simulate and verify the system-level design an executable description of the system is needed. The industry standard for executable system-level designs is SystemC, but the semantics of SystemC do not match the semantics of our formal model perfectly. This issue has been resolved by restricting SystemC to a subset of certain constructs called SystemC-PPA.

2.3 Abstract and Complex System-Level Model

As mentioned earlier, PDD bridges the gap between system-level and implementation level by generating a set of abstract properties from the system-level model and then apply refinements to it along the creation of the RTL implementation. This process, however, can be tedious in cases of complex system-level models like pipelined implementations where refining the properties can be complicated and time consuming not to mention the human error factor or the possibility of over constraining the properties.

This issue initiated the idea of splitting the system-level into two sub-levels; abstract system-level and complex system-level. These system-level models will be describing the system behaviour on two different levels of abstraction. For more understanding, we will borrow some concepts from Models of Computation (MoC) to classify these two system-level abstractions into two different models. But first, we will talk a little about the available MoCs for system-level models.

Different MoCs support different complexity and expressive power, each is tied to a level of abstracted definition of functionality, i.e. processing of data, and order, i.e. notions of time and concurrency [7].

Following the separation of computation and communication, fig. 2.2 shows the range of granularities of computation and communication for the various levels of abstraction. (A), for example, represent the behavioral system specifications as the abstract system-level model and is untimed in both computation and communication with only a causal ordering between processes. Annotating computation with execution models and estimated or measured delays results in a timed functional model (B).

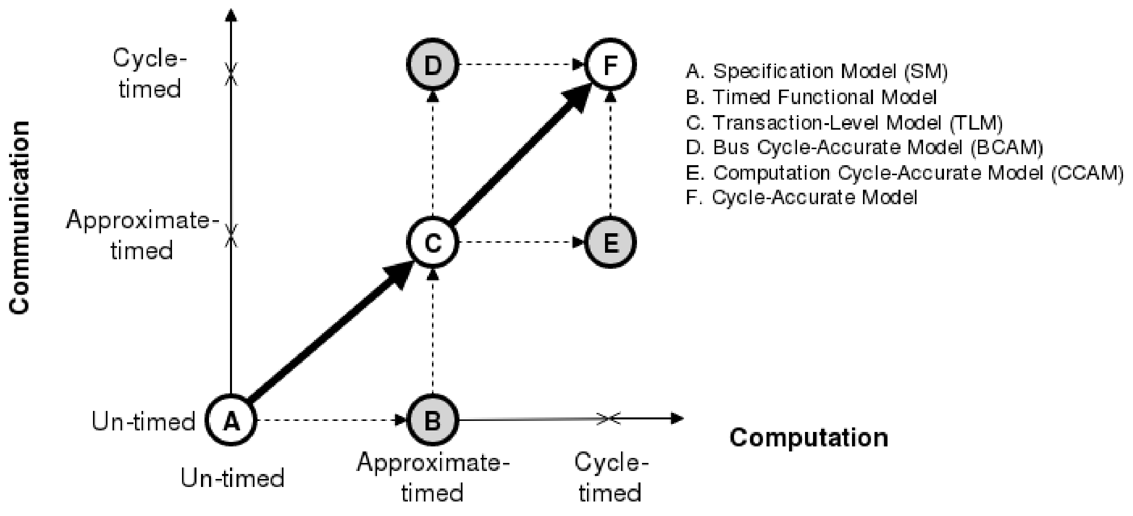

Communication
Cycle-timed
Approximate-timed
Un-timed
Computation
Un-timed
Approximate-timed
Cycle-timed
A
B
C
D
E
F
A. Specification Model (SM)
B. Timed Functional Model
C. Transaction-Level Model (TLM)
D. Bus Cycle-Accurate Model (BCAM)
E. Computation Cycle-Accurate Model (CCAM)
F. Cycle-Accurate Model

Fig. 2.2 Models granularities in PDD methodology.

Each design model is associated with a corresponding abstraction level that defines the granularity of implementation detail represented in the model. Both system-level models should enable formal methods for analysis and verification of design properties and serve as specifications for further synthesis. That's why they should both be defined with unambiguous semantics like SystemC-PPA.

2.3.1 Abstract System-Level

Abstract System-Level models are data oriented and are typically used in system behavioural models to describe desired application functionality. They represent computation as a set of concurrent processes, where processes are described in an imperative form. Definitions of concurrency and communication in such models directly translate into properties such as deadlocks and determinism, where relative ordering and interleaving of outputs and inputs is a first order property.

This level serves as the abstract golden reference model (GM) for the design and it focuses mainly on presenting the general behavior and specifications of the design. it consists of minimal amount of processing elements (referred to later as *modules*) working concurrently, each of which has its own CDFG.

2.3.2 Complex System-Level

Complex System-Level models resembles the overall topology of the system architecture and the final communication network. It contains application layer for both computation and communication. Processes are grouped under application layer of processors according to the given mapping. Application layers of communication resolves synchronization, storage and complex channels between processes down to the level of canonical primitives support by lower layers. For example, a pipelined implementation of a processor orders memory requests and memory replies differently from an abstract implementation. Also, the control unit in such an implementation can decide to flush certain input instructions and not process them.

Complex System-Level models focus on explicitly exposing and representing control flow from communication point of view. They are used for control-dominated applications and for modeling of designs at the implementation level. They explicitly represent the flow of control as transitions between different states, describing the step-wise operation of a machine in an abstracted, formalized manner. In addition, state-based models are often used to specify the abstract behaviour of control-dominated, reactive applications that are driven by actions in response to events, e.g. interrupt handling in a processor implementation. Another example of such control flow and set of states is a system-level power-aware implementation, where the model shift from a state of communicating with the environment and stacking input requests to a state of processing these requests before transitioning to a third state of producing outputs.

Complex system-level model will have a similar structure of the proposed RTL implementation including the same set of modules and the hierarchy that comes with it while still abstracting the methods of communication between these modules.

2.4 RISCV System-Level Models

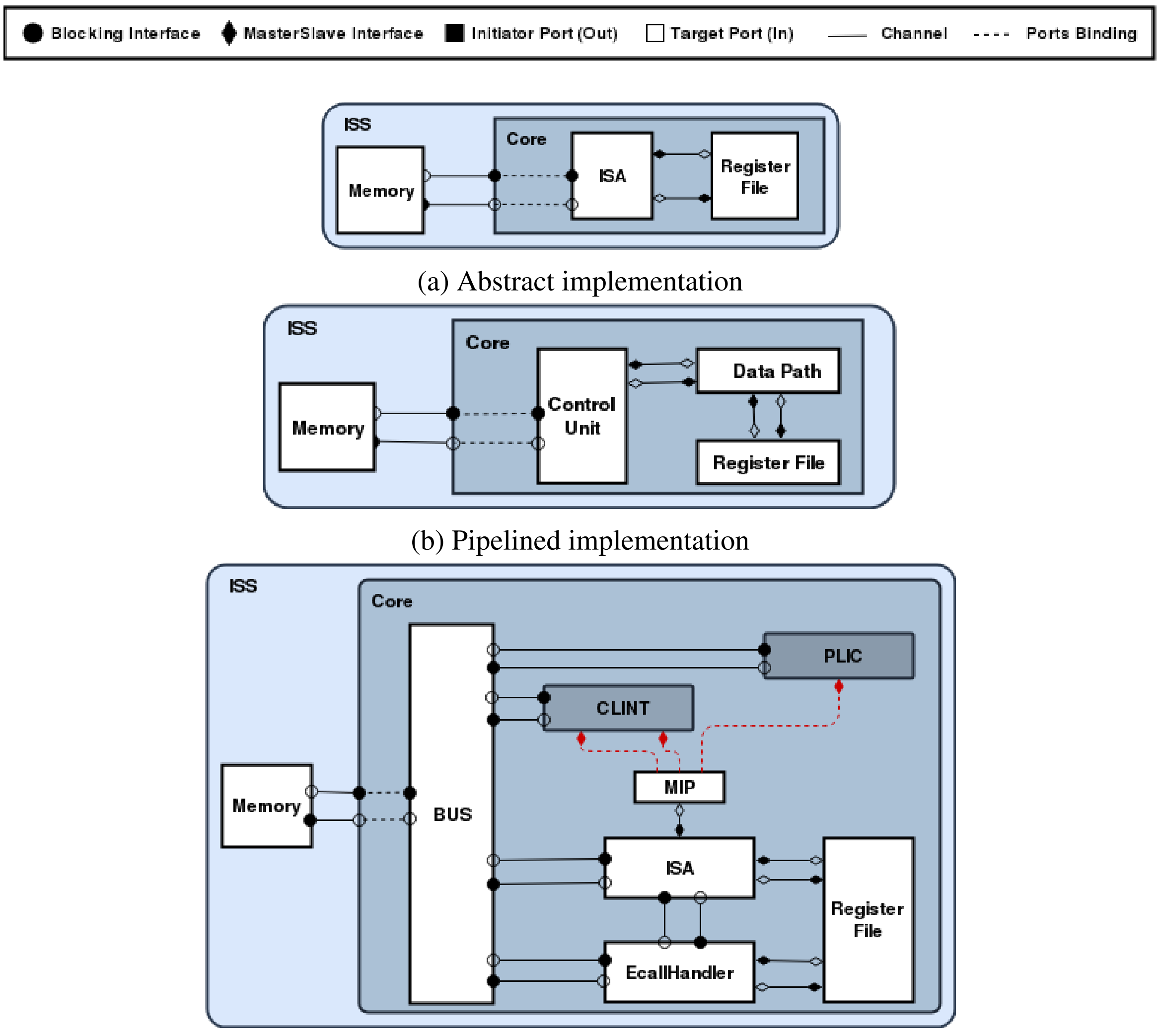
An example of a design with an abstract system-level model and more than one complex implementation is of a RISCV processor. The general architecture of RISCV processor targeted in our work is built using SystemC-PPA methodology, and the core of the RISCV processor should include one set of blocking communicating ports connecting it to the memory.

The abstract system-level of a RISCV processor core can be constructed completely within two concurrent combinational module, *Instruction Set Simulator* (ISS) and *Register File* as shown in Fig. 2.3a. It serve as a proof of concept for the RISCV specification but it doesn't represent any "real" RTL implementation. For this reason, more complex system-level models have been created to simulate a more accurate RTL implementation for such processors. One important implementation is a 5-stages pipelined processor, Fig. 2.3b, which splits the RISCV core into three modules; Control Unit, Data Path and Register File [10]. This pipelined implementation moves system-level abstraction to a lower level introducing some communication and computation functionality to the model. Another RISCV implementation was created to handle interrupts. Fig. 2.3c, while this implementation doesn't expand the execution time of instructions to 5-stages, it still creates a complex hierarchy of modules and sub-modules to manage the interrupts sources and handlers. It also expands the behavior of the processor to cover the extra specifications required for interrupt handling. More details about this RISCV implementation can be found in appendix A. All of these models are of course designed according to SystemC-PPA semantics and communicate internally through SystemC-PPA compliant channels in order to insure the ability to use them as specifications for further synthesis using SCAM tool.

2.5 System-Level Models Verification

Dividing the system-level model into two levels allows the designer to make all design decisions already at the electronic system level and takes advantage of the PPD methodology used by SCAM tool more efficiently by generating the RTL skeleton and the set of properties for each module individually. Starting from these data, the designer can focus on implementing his RTL with the minimum effort of properties refinement since each set of properties is directed to each sub-module specifically.

Thus, if the ESL design works correctly so will the RTL design implemented from it. However, as mentioned earlier, any system-level bug will appear in the RTL, and what the designer have to insure in this case, is the "*soundness*" between the abstract system-level



(a) Abstract implementation

(b) Pipelined implementation

(c) Interrupts handling implementation

Fig. 2.3 Various system-level models of a RISCV processor.

and the complex system-level. Next chapter will be dedicated to the task of verifying this soundness.

Chapter 3

System-Level Models Verification

The importance of functional correctness influences system design methodology, where in each step of the design, a designer need to make sure that the model reflects the original intent of the design and that it performs efficiently, safely and successfully. This is achieved by verification of each system design model.

The techniques for verifying design models can be classified into two groups:

- Simulation based methods
- Formal methods

In simulation based methods, the specification is a set of properties that the implementation model must be checked for. Some instances of these properties are expressed as pairs of stimulus and expected behaviour. The stimulus forms the input to the implementation model being simulated and the expected behaviour is checked by monitoring the output of the simulated model, fig. 3.1.

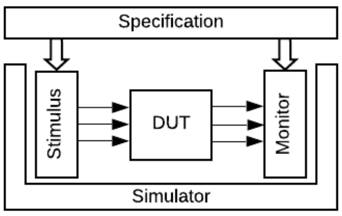


Fig. 3.1 A typical simulation environment.

In formal verification methods, a property is statically checked instead of some instances of the property. This means that once the verification process is complete, we can be assured that the implementation model satisfies the property under all inputs.

At first sight, simulation may seem too expensive, too time consuming or even less trustworthy than formal methods. Indeed, simulation is only a partial test since we are checking for instances of a property and not the complete property under all input scenarios. However, simulation is still predominant technique of verification for various historical as well as practical reasons [7]. Among which is the lack of formal verification methods for system-level, where model complexity is still manageable.

As designs become larger and more complicated, simulation takes far too long to meet the required verification quality. As a result, techniques like assertion based verification are being used to complement the traditional simulation and debugging of design model.

In our work, we will focus on verifying the functional correctness of the complex system-level model compared to the abstract model according to *Coverage Driven Verification* (CDV) methods, which consists of automated stimulus generation, independent result checking and coverage collection. We will be using Universal Verification Methodology (UVM) techniques applied on SystemC for this end.

In this chapter we will talk more about UVM and the adaptation of it for two different examples, while next chapter will talk about CDV in more details.

3.1 Verification of System-Level Models

As the design abstraction level rises, system level models are being used increasingly for validation. During system level design, these models are refined into more detailed models to resemble more and more the logic level model. As a result, we are faced with the problem of verifying the equivalence of abstract system level models and the refined lower levels of abstractions.

In its sense, PDD methodology can be observed as a model refinement based design methodology, where each model produced by refinement is equivalent to the input model. An example of that is the relationship between the abstract and complex system-level models. The notion of equivalence comes from the simulation semantics of the model. Two models are equivalent if they have the same simulation results.

Designer decisions are used to add details to a model to refine it to the next lower level of abstraction. Each designer decision corresponds to several computation and communication objects or replace an object in the model with one from the library. However, correct refinement doesn't mean that the output model is bug free. That's why we still need to use traditional verification techniques on the system-level models and prove the equivalence of objects that can be replaced by one another.

3.2 Universal Verification Methodology

UVM is used to create modular, scalable, configurable and reusable testbenches based on verification components with standardized interfaces. it's also an environment supporting migration from directed testing towards Coverage Driven Verification (CDV). However, the verification engineer is responsible for coverage and randomization definitions; UVM only offers the hooks and technology (i.e. classes) [18]. UVM, in its origin, is a library of SystemVerilog classes, but recently an adaptation of this library at SystemC has been introduced to elevate verification towards system-level.

UVM works best in Bus-Based, High-Traffic System-Level designs. And, it's most efficient when verification procedure requires randomization and/or functional coverage. It can still be adapted to handle other classes of designs but may not be the best solution in those cases. UVM uses mainly black-box verification approach, so it doesn't check the internals of the DUT [12].

The main concepts of UVM can be summarized in the following:

- Clear **separation** of test stimuli (sequences) and test bench: Sequences are treated as "transient objects" and thus independent from the test bench construction and composition. In this way, sequences can be developed and reused independently.
- Introducing test bench **abstraction levels**: Communication between test bench components based on transaction level modeling (TLM).
- Reusable verification components based on standardized interfaces and responsibilities. such *Universal Verification Components* (UVCs) offer sequencer, driver and monitor functionality with clearly defined (TLM) interfaces.
- Non-intrusive test bench **configuration** and **customization**. where hierarchy independent configuration and resource database to store and retrieve properties are used everywhere in the environment. And factory design pattern introduced to easily replace UVM components or objects for specific tests.
- Well defined **execution** and **synchronization** process: Simulation based on phasing concept, build, connect, run, extract, check and report. As well as, objection and event mechanism to manage phase transitions.

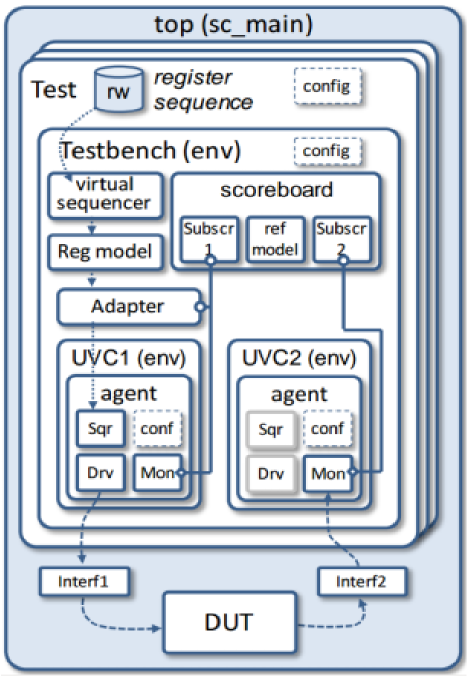


Fig. 3.2 UVM general architecture.

3.2.1 UVM general Architecture

Figure 3.2 illustrate how the general architecture of a UVM test suite looks like.

- The top-level (e.g. sc_main) contains the test(s), the DUT and its interfaces.
- The DUT interfaces are stored in a configuration database, so it can be used by the UVCs to connect to the DUT.
- The test bench contains the UVCs, register model, adapter, scoreboard and (virtual) sequencer to execute the stimuli and check the result.
- The test to be executed is either defined by the test class instantiation or by the member function run_test.

3.3 UVM-SystemC structure with Bus implementations

A UVM-SystemC testing suite was created for testing various implementations of a Bus design. Every implementation should have similar communication ports with the environment to the ones used in the *Golden Model* (GM). For this test suite, we are testing designs that connect one master to four slaves as shown in listing 3.1.

Listing 3.1 Bus interface with the environment.

```
//In-port
blocking_in<bus_req_t> master_in;
blocking_in<bus_resp_t> slave_in0;
.......
//Out-port
blocking_out<bus_resp_t> master_out;
blocking_out<bus_req_t> slave_out0;
.......
```

The main idea of testing such implementations is by sending a sequence of random requests from the master (on the *master_in* port) and checking the series of requests forwarded towered each slave (on *slave_outx* ports). And from the other side sending random responses from the slaves (on *slave_inx* ports) and checking the series of responses forwarded towered the master (on *master_out* port).

However, designers can creating their implementations in different ways, adding features and complexities to the abstract design. This can affect the designs interaction with the environment and result in a different behaviour from the abstract model behaviour. Thus, we classify these implementations according to their communication with the environment into three categories that we'll explain in more details afterwords.

- Orderly implementations.
- Disorderly implementations.
- Interactive implementations.

3.3.1 Structure for orderly implementations

We refer here to implementations that have one-to-one behavior between inputs and outputs as **orderly** implementations. The meaning of one-to-one behavior is that the DUT reacts to input; i.e. provide an output, sequentially before asking for a new input. These implementations behavior resembles the golden model behavior. For this reason, it's possible to compare the outputs of the DUT instantly with the outputs of the GM.

As a result, we have designed the testing suite with five *active* UVCs. One UVC *drives* master's requests and *monitors* it's responses while each of the other four UVCs *monitors* a slave's requests and *drives* it's responses for each slave respectively. All *driven* sequences (master's requests and slave's responses) need to be exported to the scoreboard in order to run the same scenario on the golden model. At the same time, all *monitored* outputs to the environment (slave's requests and master's responses) will be exported to the scoreboard as well in order to compare with golden model's outputs. The designed UVM-SystemC test

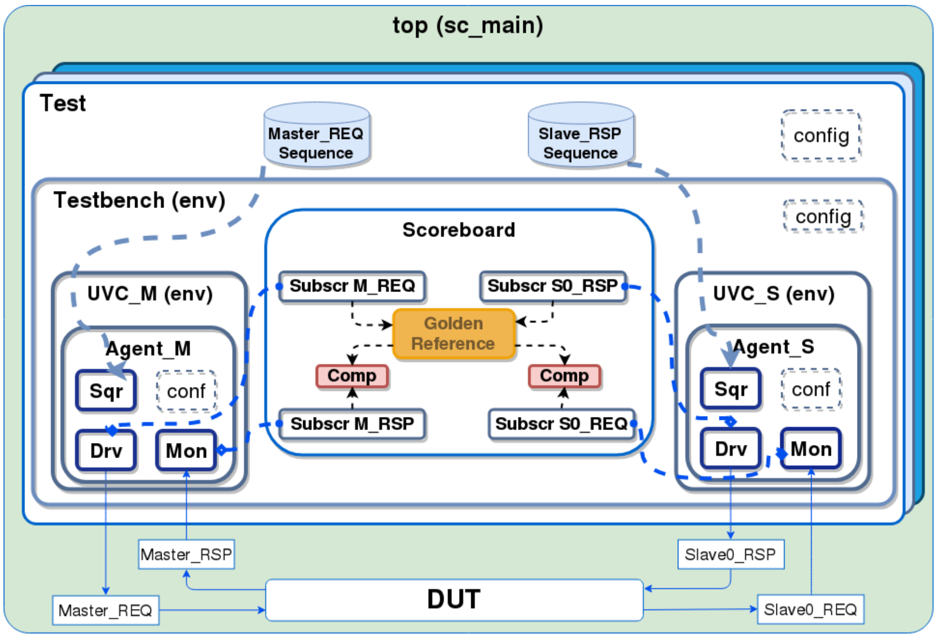
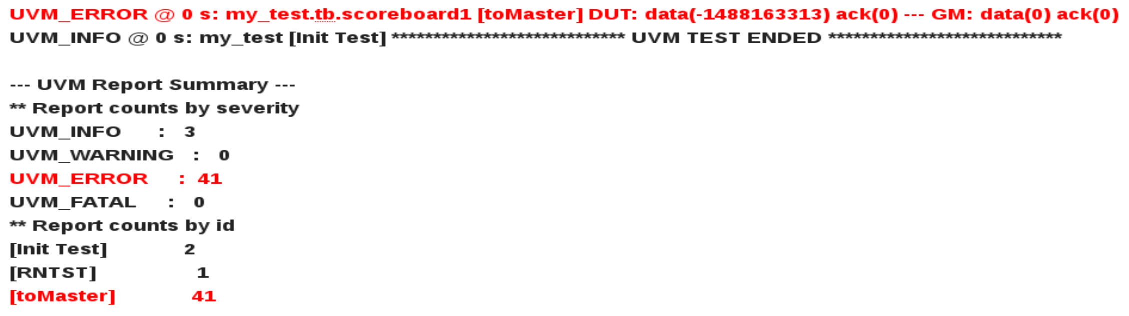


Fig. 3.3 UVM architecture for an orderly Bus implementation with 1 Slave.

suite for such bus implementations can be seen in figure 3.3, adding new slaves ports to the DUT require adding UVCs for each slave with the associated subscribers at the scoreboard.

The main implementations in question for this testing suite was of a Wishbone bus, and through simulation with a sequence of 100 master requests, with random data and request type, we managed to find some inconsistency in master responses between the GM and the DUT. This inconsistency is noted in the report provided by the UVM testing suite at the end of the simulation as shown in figure 3.4.


```
UVM_ERROR @ 0 s: my_test.tb.scoreboard1 [toMaster] DUT: data(-1488163313) ack(0) --- GM: data(0) ack(0)
UVM_INFO @ 0 s: my_test [Init Test] ***************************** UVM TEST ENDED *****************************

--- UVM Report Summary ---
** Report counts by severity
UVM_INFO     :  3
UVM_WARNING  :  0
UVM_ERROR    :  41
UVM_FATAL    :  0
** Report counts by id
[Init Test]        2
[RNTST]            1
[toMaster]        41
```

Fig. 3.4 UVM-SystemC report showing inconsistency between GM and DUT.

Through debugging, we managed to find the source of inconsistency in the Wishbone implementation and fix it.

3.3.2 Structure for disorderly implementations

As opposed to orderly one, a **disorderly** implementation might request more than one input before providing any outputs. such a case can occur in pipelined or power-aware implementations. These implementations behave differently from the golden model regarding the sequence of inputs and outputs which makes it impossible to compare the outputs of the DUT instantly with the ones of the GM.

For such implementations we modify the Scoreboard in the UVM-SystemC test suite to separate the execution of GM from the execution of the DUT, figure 3.5. In this case, we store every communication toward and from the DUT (provided by the Subscribers) in *vectors* and run the inputs on the GM separately while storing GM outputs as well in compliant *vectors*. While along the run of the simulation, a comparison will be running on the outputs of both GM and DUT, when available, to assert equal behavior of both implementations.

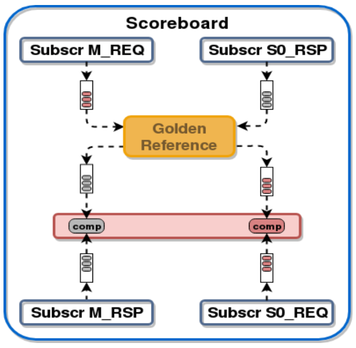


Fig. 3.5 UVM-SystemC Scoreboard with separate execution of GM and DUT for disorderly implementations.

3.3.3 Structure for interactive implementations

Another type of implementations can interact with the testing environment, like requiring to repeat a request for a Bus implementation or jump to a different address in a program for processors. such implementations might differ from the GM in the way of handling the test stimuli (i.e. sequence). For such cases, we move further in the separation of the two models execution to the level where we duplicate the stimuli and run both on the GM and the DUT separately, figure 3.6.

However, for such implementations, asserting the equivalence of behavior to the GM becomes *selective*. An example of such situation is when the Bus implementation responses to a master request with a *Retry* acknowledgement because it's not ready to handle the request at that moment, while the abstract GM will never respond with such acknowledgement. In

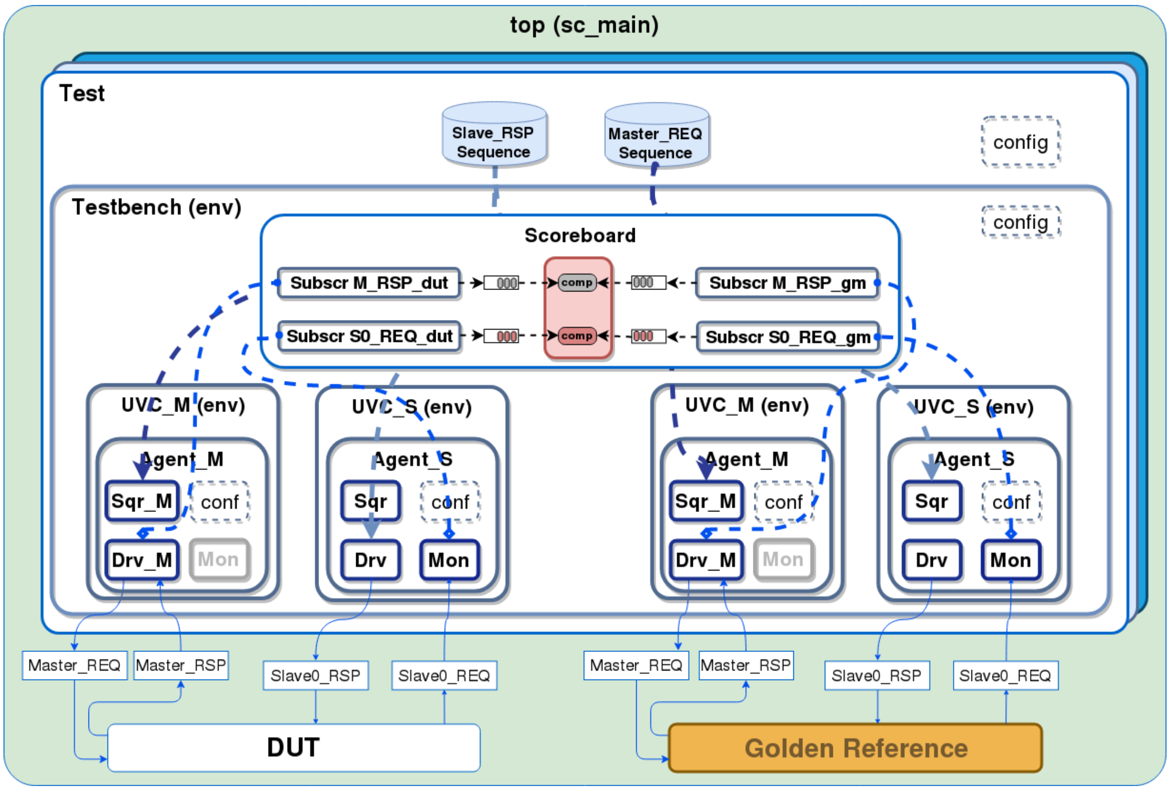


Fig. 3.6 UVM architecture for an interactive Bus implementation with 1 Slaves.

this situation, the assertion of functional behavior will discard such responses and compare only the *important* responses (with *OK* acknowledgement) which represent the desired behaviour to be tested.

3.4 UVM-SystemC structure with RISCV

The general architectures of RISCV processors targeted with our test suite are built using SystemC-PPA semantics as explained in previous chapter. The core of the RISCV processor should include one set of communicating ports connecting it to the memory (Listing 3.2).

Listing 3.2 RISCV core interface to the memory.

```
// Ports (Memory Interface)
blocking_out<CUtoME_IF> COtoME_port;
blocking_in<MEtoCU_IF> MEtoCO_port;
```

RISCV processors should classify under the interactive implementations and since GM and DUT run on different speeds we split testing by loading a copy of the program (i.e. test stimuli) to each design. While results comparison is handled by buffering the outputs of both models in the scoreboard and executing the comparisons when new outputs are available from both models [14]. Like mentioned earlier about interactive implementations, asserting behaviour equivalence of GM and DUT is *selective*. Since that our only output interface of the processor is the memory interface, what we can assert in this case is the *memory write* outputs coming from store instructions. We can assert in this case the correct behaviour regarding how the processor is affecting the environment (i.e. memory) with its writing commands.

Giving the nature of the design and the type of interfaces connecting it to the environment, some changes were made to the general architecture of the UVM-SystemC testing suite.

Interactive sequences

When attempting to run a program test on the DUT it's important to feed the DUT sequentially valid instructions and data. This means that the sequence containing the program-to-run should be able to send to the DUT the right instruction (on the *MEtoCO_port* interface) from the right address that the DUT requested (on the *COtoME_port* interface). To manage this issue we use the driver's reply to each sequence item as a new instruction request coming from the DUT. And this will require that the same drive will also have the *COtoME_port* DUT interface and of course its corresponding *item_collected_port* to the scoreboard.

Keeping track of the internal status of the DUT

UVM-SystemC treats the DUT as a black-box, preventing access to the internal registers of the DUT. In order to verify the values held in these registers during the simulation run, we take a *snapshot* of the DUT internals. We keep track of the internal status of the DUT (The register file), by periodically intercepting the program sequence running on the DUT –This period can be defined by a number of instructions interval, *CHECK_INTERVAL*– and sending a short sequence of instructions storing the registers values to memory. This short sequence will give the scoreboard the chance to compare the internal status of the DUT with the internal status of the golden reference. Unfortunately, this technique doesn't work properly on pipelined implementation because of the complexity of handling instructions in this implementation. The fact that instructions can be stalled or flushed makes it complicated to control such interception of the program.

It is worth noting, however, that this technique will require assurance that the store instruction works properly, among other types of instructions. That's why dedicated tests should run first on the DUT targeting each instruction type before we move to a more complicated tests.

Different tests with different scenarios

As mentioned earlier, the verification engineer should insure that all instruction types work properly on the processor implementation before executing more complicated tests. For this purpose, we created different test scenarios to verify each instruction type individually. Another test scenario was created as well to allow the user to select a pre-written program to run on the DUT. Nevertheless, each test scenario should of course end with a *snapshot* sequence of instructions that will give the scoreboard the chance to verify the internal status (i.e. registers values) of the implementation. Since UVM support separation of tests and test bench, it became fairly easy to create separate tests while maintaining the same topology of the test suite. Afterwards, desired tests can be selected (passed) as command line options.

UVM-SystemC for RISCV Layout

As a final result to the previously mentioned issues and others, we came out with a UVM-SystemC test suite that can be seen in figure 3.7.

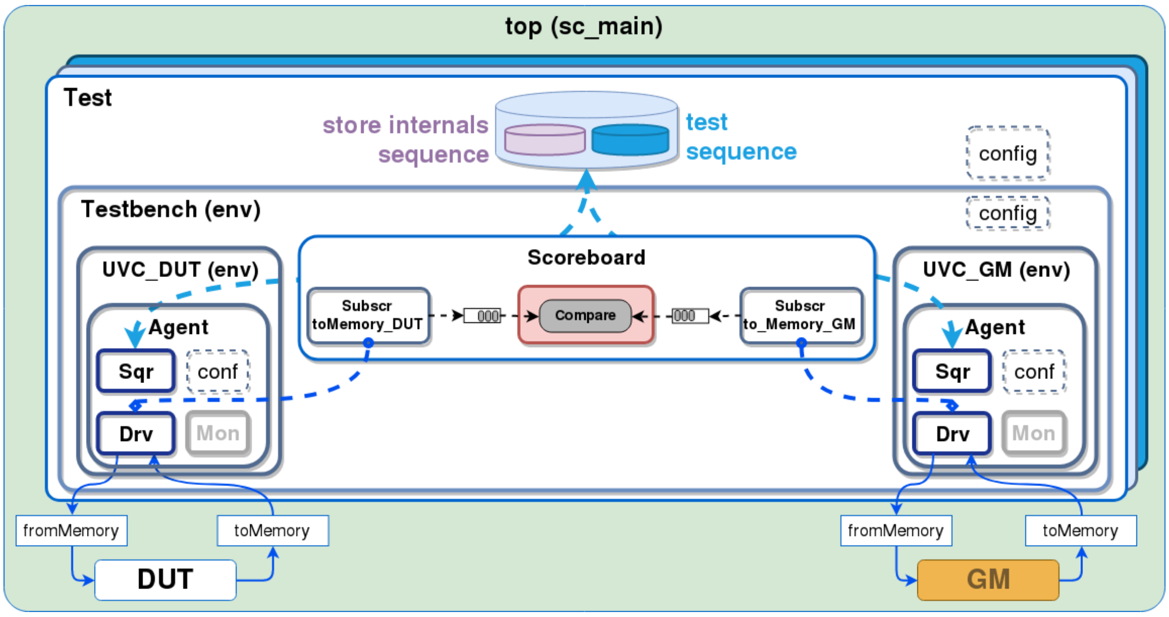


Fig. 3.7 UVM architecture for a RISCV core following SystemC-PPA.

Chapter 4

Simulation Coverage

Coverage-driven verification is a verification methodology that uses coverage metrics to help quantify verification completeness, and identify inadequately exercised design aspects and guide input stimulus generation. Coverage metrics are a set of numbers used to determine how much design functionality the verification suite has exercised. Functional metrics and structural metrics are the two major classifications in verification metrics [19].

The importance of coverage-driven approach comes from the impracticality of using verification flow designed for abstract implementation to a more complex implementation. Consider, for example, the pipelined implementation of the RISCV processor in our study. Applying the same testing method of the abstract system-level model to this implementation must produce the same results. However, the architecture of the pipelined implementation was invented to meet more stringent performance requirements, which will come with a set of boundary conditions and bugs. Hence, a new verification problem. Unless the new verification problem is described, quantified and implemented using a coverage-driven approach, the probability of taping out the design with latent bugs is high [17].

coverage is used as a confidence-building metric, ensuring that the design was verified as thoroughly as possible [11]. Verification engineers spend a considerable amount of time developing test suites in order to achieve relatively high coverage metrics that will indicate how effective these test suites are at verifying the functionality of the design. The remainder of their task is spent either tweaking the test suite to increase the coverage numbers or justifying why some of these numbers cannot—or need not—reach 100%.

In this chapter, we will talk about some of the main coverage metrics available and the approach used to combine coverage metrics in our verification methodology mentioned in previous chapter.

4.1 Structural Coverage

Structural metrics refers to metrics that create reports on structural issues, such as which lines of code or which states in a state machine have been exercised. This class has a weaker correlation to functionality defects than functional coverage class. For example, if an entire set of tests were executed, and findings show that lines of code used to implement the multiply instruction of a processor weren't executed, further investigation is required to determine if a defect was found. Structural metrics, however, are easier to implement because they require no additional effort by the verification engineer except execution of a code coverage tool [8].

Most design teams focus on structural metrics, such as code coverage, because it is easier to implement and interpret. The simplest structure to target is of a line or block of code (a sequence of lines with no control branches) separated by control statements which constitute the branching points in the Control Flow or FSM of the design. Measuring code coverage requires little overhead, and because of the ease of interpreting coverage results, these metrics are useful for writing test cases to improve coverage.

Coverage flags can be sub-divided further on system-level model into control structure flags measuring coverage of the transitions in the FSM of design's modules and circuit structure flags measuring coverage of the various return options of the constant functions used by these modules. For example, in the ISA module of the system-level design of RISCV processor we use, for some instructions, a constant function to read a source register. While the FSM of the module doesn't care about which register is being read, CDV should cover all the options or paths available in this function.

4.1.1 Structural Coverage Generation using SCAM

Tools that use code coverage metrics instrument the design with testing suites and captures data during simulation runs, and then the resulting data is accumulated and analyzed. This kind of technology treats the design as a *white-box* in order to measure code coverage during the simulation run [21]. However, in our verification environment, i.e. UVM-SYSTEMC, we are treating the design as a black-box. We solved this contradiction by presenting a feature to SCAM tool that re-write the system-level model of the DUT with coverage flags following every branching point, Fig. 4.1.

This process starts with the SystemC-PPA model of the design. SCAM tool parses the model, using LLVM/Clang compiler, and creates the *Abstract Syntax Tree* (AST) which contains the model code and all static information related to it in a software design pattern called visitor pattern. SCAM further analyzes the AST to strip away irrelevant information and keep only the information describing the module in a new data structure called Abstract

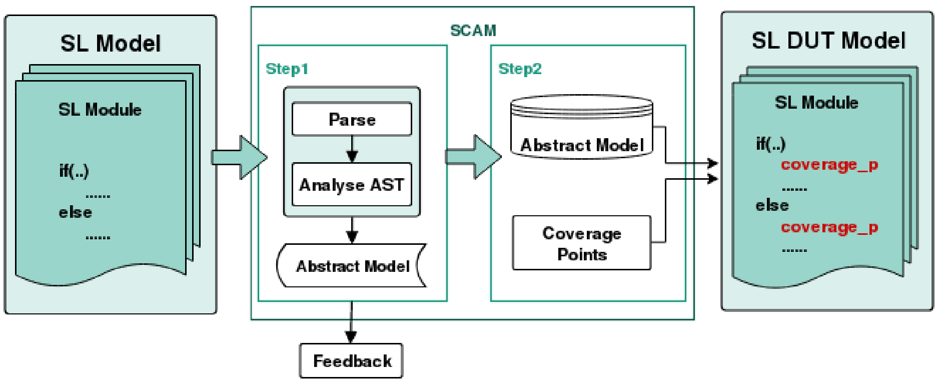
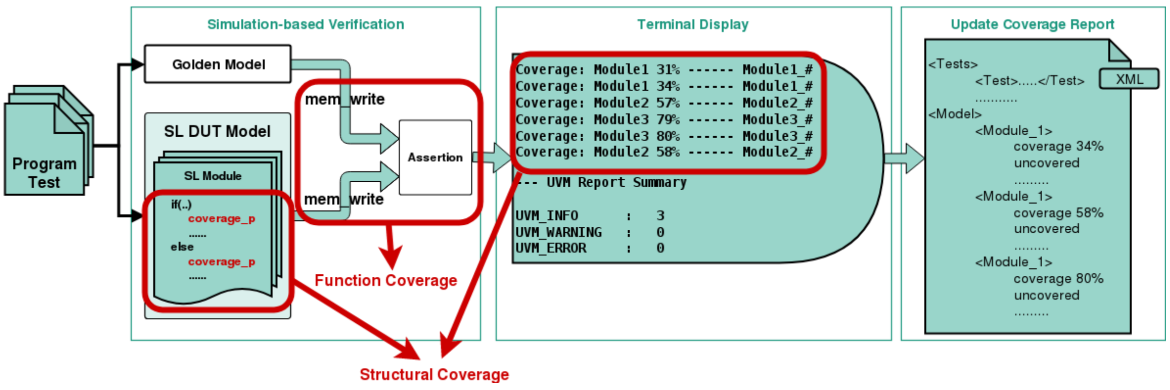


Fig. 4.1 SCAM process of re-writing the system model with coverage points

Model (AM). The added feature re-write the AM code while inserting a coverage point flag when visiting every branch of every *if-then-else (ITE)* structure of the model.

4.1.2 Coverage Report for SL model Verification

By using the re-written system-level model of the design in the verification test suite, we get to explore the efficiency of our given tests in covering the design while verifying at the same time the behaviour of the design. We measure this efficiency by presenting the coverage status of the design in an XML style report. This report indicate, for each sub-module, the coverage percentage achieved with simulation and a list of all the coverage points that haven't been covered yet, Fig. 4.2.



```
Coverage: Module1 31% ------ Module1_#
Coverage: Module1 34% ------ Module1_#
Coverage: Module2 57% ------ Module2_#
Coverage: Module3 79% ------ Module3_#
Coverage: Module3 80% ------ Module3_#
Coverage: Module2 58% ------ Module2_#
--- UVM Report Summary

UVM_INFO       :   3
UVM_WARNING    :   0
UVM_ERROR      :   0
```

```
<Tests>
        <Test>.....</Test>
        ...........
<Model>
        <Module_1>
              coverage 34%
              uncovered
              .........
        <Module_1>
              coverage 58%
              uncovered
              .........
        <Module_1>
              coverage 80%
              uncovered
              .........
```

Fig. 4.2 Model verification and coverage reporting process.

After every test executed on the re-written DUT, the report can be updated to present a list of all executed tests and the added coverage gained from them. It also updates for each module the maximum coverage percentage achieved from all the tests and the list of remaining uncovered points. Listing 4.1 shows an example of such a report for RISCV pipelined implementation after running a set of pre-written tests we had available.

Listing 4.1 Coverage report for simulation-based verification of a RISCV pipelined implementation.

```
<UVMSYSTEMC_Report>
    <Tests>
        .......
        <Test Type="Program">
            <Path>test_1.elf</Path>
            <Coverage_Percent name="Control_unit">59</Coverage_Percent>
            <Added_CoveragePoints name="Control_unit">1</Added_CoveragePoints>
            <Coverage_Percent name="Data_path">86</Coverage_Percent>
            <Added_CoveragePoints name="Data_path">8</Added_CoveragePoints>
            <Coverage_Percent name="Register_file">100</Coverage_Percent>
            <Added_CoveragePoints name="Register_file">1</Added_CoveragePoints></Test>
        .......
    </Tests>
    <DUT_Core>
        <Module name="Control_unit">
            <Coverage_Total>318</Coverage_Total>
            <Coverage_Percent>75</Coverage_Percent>
            <Coverage_Points>
                .......
            <Uncovered_Points>
                <Point name="Control_unit_6">false</Point>
                <Point name="Control_unit_20">false</Point>
                <Point name="Control_unit_25">false</Point>
                <Point name="Control_unit_42">false</Point>
                <Point name="Control_unit_47">false</Point>
                <Point name="Control_unit_89">false</Point>
                <Point name="Control_unit_97">false</Point>
                <Point name="Control_unit_120">false</Point>
                <Point name="Control_unit_146">false</Point>
                <Point name="Control_unit_160">false</Point>
                <Point name="Control_unit_165">false</Point>
                <Point name="Control_unit_182">false</Point>
                <Point name="Control_unit_187">false</Point>
                <Point name="Control_unit_199">false</Point>
                <Point name="Control_unit_209">false</Point>
                <Point name="Control_unit_223">false</Point>
                <Point name="Control_unit_233">false</Point>
            </Uncovered_Points>
        </Module>
        <Module name="Data_path">
            <Coverage_Total>139</Coverage_Total>
            <Coverage_Percent>86</Coverage_Percent>
            <Coverage_Points>
                .......
            <Uncovered_Points>
                <Point name="Data_path_88">false</Point>
                <Point name="Data_path_133">false</Point>
            </Uncovered_Points>
        </Module>
```

```
50          <Module name="Register_file">
51              <Coverage_Total>63</Coverage_Total>
52              <Coverage_Percent>100</Coverage_Percent>
53              <Coverage_Points>
54                  .......
55              <Uncovered_Points>
56                  </Uncovered_Points>
57          </Module>
58      </DUT_Core>
59  </UVMSYSTEMC_Report>
```

Through analyzing this report after each test run, the designer can get an idea about what branches in the design code haven't been triggered yet and can create more testing scenarios to cover these branches. Line 46, for example, shows a coverage point that hasn't been trigger (`Data_path_88`). By investigating the location of this point in the re-written code of the design, shown in listing 4.2, we find that branching point, in line 4, indicates that no test used register x22 as a source register. A new test should be written that includes an instruction to test such case.

Listing 4.2 Re-written Data_path module to include coverage points.

```
1  unsigned getRegContent(RFtoDP_IF RFtoDP_data, RF_RD_AccessType rdReq, unsigned srcAddr) {
2      if ((rdReq == RF_RD)) {
3              .......
4          if ((srcAddr == 22)) {
5              if(!coveragePoints["Data_path_88"]){
6                  .......
7              return(RFtoDP_data.reg_file_22);
8          } else {
9              .......
```

4.2 Functional coverage

While structural coverage metrics focus mainly on the code and FSM of the design, functional metrics target the functionality exercised in the design. For example, a defect is found if required functionality, such as the execution of a multiply instruction in a processor, is exercised and it fails. These functional metrics provide as well a functional view of verification completeness, which is more desirable than a structural view. However, functional metrics are difficult to implement because the verification engineer must extract the list of functionality to be searched for, and devise means of proving that the functionality was exercised.

Many engineers perform manual functional coverage which involves manually reviewing the results of a test to determine the functionality exercised. Manual review can mean looking at signals exchanged in communication between modules or the model and the environment, like checking every memory access request from processor to memory. This time-consuming process usually occurs only a small number of times or even just once, because the functionality should be defined early enough in the design process. Thus, the method of achieving functional coverage doesn't change with bug fixes or features additions during the development cycle.

4.2.1 Functional Coverage using UVM-SystemC

Functional verification is best exercised on system-level model which is supposed to represent the specifications and desired functionality of the design. With our approach, we push verification to the complex system-level implementation using UVM-SystemC methodology as our verification environment.

Since this methodology treats the design as a black-box, we focus mainly on verifying functionality of the design by observing its interaction with the environment through the communication ports of the design. Hence, assessing functional coverage can be done using *Operational Assertions*, assertions that express functionality at the specification level. We apply these assertions on the monitored outputs of both abstract and complex system-level models at the scoreboard, assuming of course the functional correctness of the abstract system-level. If both models produce the same output then they are functionally equivalent.

Adding feature like pipelining to the design can affect the designs interaction with the environment. Such cases require re-evaluating what functionalities to be tested in the verification process. We encountered this when attempting to verify RISCV pipelined design using UVM-SystemC test suite. The RISCV models have only one pair of communication port with the environment, the memory access ports, as mentioned earlier in Listing 3.2. Asserting every memory request from the processor doesn't make sense in this case because pipelined implementation might request to read instructions and then flush them. Hence, functional assertion have to target more specific interaction with the memory which is the *memory write* requests in particular as shown in Fig. 4.2. This behaviour justify the selective assertion of interactive implementations mentioned earlier in section 3.3.3.

4.3 Performance Improvements

In an ideal scenario, one would like to run the minimum number of test cases to cover as much of the design as possible within the shortest time possible. For this end, a few improvements can be made to increase the efficiency of simulation-based verification.

Random Stimulus

The set of tests executed in the example mentioned earlier in listing 4.1 come from pre-written scenarios specified explicitly to verify certain functionalities of the design. Simulation with this kind of testing scenarios is considered as *direct simulation* which has the advantage of high execution speed. However, direct simulation only checks single or specific scenarios and writing such scenarios manually is a very time-consuming task [16].

To overcome this limitation, *random pattern simulation* is used to generate random stimulus pattern for the design. In our verification methodology, we use SystemC Verification (SCV) library to generate randomize stimulus patterns for the design. One of the appealing features of this library is the ability to generate random data for compound type variables.

Constrained Random Stimulus

The simulation performance can be improved by choosing test cases intelligently to maximize coverage with minimal simulation runs. One optimization is to reduce test generation time by giving constrains to stimuli and testing with only valid inputs. Since the design is typically constrained to work for only selected scenarios, we can use this knowledge to generate only the valid test vectors.

Another reason for our choice of SCV library was the ability to control the process of generating random stimulus in order to produce *constrained random stimulus* that represent valid input to test the functionality of the design. An example of such constraining is for *J_Type* instructions for RISCV processor shown in Listing 4.3.

Listing 4.3 Constraining *J_Type* instructions for RISCV processor.

```
SCV_CONSTRAINT_CTOR(J_Type) {
        decodedInstr->instrType.keep_only(InstrType::G_INSTR_JAL);
        decodedInstr->encType.keep_only(EncType::G_ENC_J);
        SCV_CONSTRAINT(decodedInstr->rs1_addr() = 0);//doesn't have an effect
        SCV_CONSTRAINT(decodedInstr->rs2_addr() = 0);//doesn't have an effect
        SCV_CONSTRAINT(decodedInstr->rd_addr() < 32);
        SCV_CONSTRAINT(decodedInstr->imm() <= 0xFFFFF);
    }
```

Monitor Optimization

Monitoring only the primary output of the design during simulation lets us know if a bug exists without having to look deeper inside the design, especially because we treat the design as a black-box. However, when the source code of the model is available and some bugs were found, assertions can be placed on internal variables or signals in the model to locate more accurately where the bugs are.

4.4 Challenges

4.4.1 FSM Path Transition Coverage

Code coverage metrics are defined on static, structural representations; hence their ability to exploit sequential behavior is limited. Metrics defined on state transition graphs are more powerful in this regard. These metrics require state, transition, or limited path coverage on FSM system representation, this *pair-arcs* metric requires exercising all feasible pairs of transitions for each pair of modules FSMs. Because FSM descriptions for complete systems are prohibitively large, these metrics must be defined on smaller, more abstract FSMs. The separation of constant functions from FSM in SystemC-PPA semantics allows for such abstract FSMs to be designed, but with more complex designs containing deep hierarchical modules, attaining high coverage for such metrics becomes difficult. Also, designers may need to consider paths rather than only states or transitions to ensure that important sequences of behavior are exercised. Some automated approaches involve sequential testing techniques, others establish a correspondence between coverage data and input stimuli using pattern matching on previous simulation runs. Nevertheless, state-space-based metrics are invaluable for identifying rare, error-prone execution fragments and FSM interactions that may be overlooked during simulation, thus justifying the high cost of test generation.

4.4.2 Eliminating False Negatives - Reachability

After achieving relatively high coverage metrics, whether with user developed test suites or random stimulus, verification engineers have to justify why some of these numbers cannot—or need not—reach 100%. An example of such case can be seen in listing 4.1, where coverage point `Control_unit_146` covers the conditional branch in line 9 of listing 4.4. Such coverage point will never be covered because no valid input stimulus will reach this branch. Thus, coverage report will always give less percentage than what actually been covered, this inadequacy is nothing but *false negative*. Eliminating such false negatives is one of the greatest challenges facing automated tools.

Listing 4.4 Part of Control_Unit code showing an unreachable branch.

```
1  InstrType Control_unit::getInstrType(unsigned int encodedInstr) const {
2      if (OPCODE_FIELD(encodedInstr) == OPCODE_R) {
3          if (FUNCT3_FIELD(encodedInstr) == 0x00) {
        if (FUNCT7_FIELD(encodedInstr) == 0x00) {
            return InstrType::INSTR_ADD;
        } else if (FUNCT7_FIELD(encodedInstr) == 0x20) {
            return InstrType::INSTR_SUB;
        } else {
            if(!coveragePoints["Control_unit_146"]){
                ......
            return InstrType::INSTR_UNKNOWN;
        }
```

4.4.3 Observability

Standard structural coverage metrics measure only control coverage, that is, the degree to which stimuli exercises the code. Control coverage indicates which parts of the code have been exercised and helps to answer the question, *"Have I written enough stimuli?"* [6]. But when a stimulus has triggered a fault, how do we know whether the fault has been observed by a checker? Observation coverage identifies which parts of the code have been checked and which parts have not, and helps to answer the question, *"Have I written enough checks?"*. Thus, while control coverage is necessary, it's not sufficient to ensure observation coverage.

In a simulation-based validation, the implementation model's variables that are checked against required behavior in this manner are observed variables. while the correct behavior of the remaining variables is not specified. Hence, they are called unobserved variables. A discrepancy from desired behavior is detected during simulation only if an observed variable takes on a value that conflicts with the value specified by the reference model.

The observability requirement does not mean that a portion of the design must behave incorrectly during simulation or that the design must be simulated with an artificially injected error. The only requirement is that the portion must affect the value of an observable variable in some clock cycle. Thus, designers are assured that had there been an error in that portion, the framework would have detected it. An unobservable error is one that no simulation run can detect. If all variables that are crucial to the design's functionality are observable, then in the worst case, unobservable errors will cause performance penalties but not incorrect behavior. An unobserved error, however, could propagate to an observable variable during a simulation run. The fact that we have excited but not observed the error merely indicates that we have not driven the design with the appropriate inputs to propagate the error to an observed variable.

Since we are applying CDV on a re-written DUT using SCAM tool, we can improve SCAM tool to get rid of unobservable variables and ensure that there are no unobservable variables in the final design.

Chapter 5

Summary

Until this day, industry hasn't adopted formal verification completely, especially when it comes to abstraction model levels, and simulation-based verification is still in use. However, new challenges to simulation-based verification result from the growth in size and complexity of designs. The sheer size of designs makes simulation of RTL model too expensive and time consuming. To answer this challenge, we must develop a comprehensive and formal system level methodology which will require formal semantics for different system level models. As a result, traditional simulation based verification methods can still be used for system specification model while correct refinements will avoid the need to simulate lower RTL models. Meanwhile, tools like SCAM that insure a *complete* set of properties, can boost confidence in formal verification coverage on RTL models. Well defined model semantics would make it possible to define and prove correct transformation for automatic model refinement. Therefore, model formalization would make complete system verification much faster.

Combining both steps of verification in a top-down design methodology like PDD is possible when formal semantics like System-PPA is used. Using such semantics will help defining formally the required *testing points* of the design and how to reach them [4]. Simulation-based verification asserts the behaviour of the complex system-level model in reference to the abstract system-level model, this step will be covering the main testing points concerning specifications and functionality of the design. While on the other hand, formal verification evaluates RTL behaviour according to operational properties coming from the complex system-level model, insuring coverage of all complex corner cases and test points of the design.

References

[1] Amazon Web Services (AWS) (2019). The FreeRTOS™ Kernel. [online] https://www.freertos.org/RTOS.html.

[2] Andrew Waterman, K. A. (2017a). The RISC-V instruction set manual, volume I: User-level ISA. [online] https://riscv.org/specifications/.

[3] Andrew Waterman, K. A. (2017b). The RISC-V instruction set manual, volume II: Privileged architecture. [online] https://riscv.org/specifications/privileged-isa/.

[4] Aritra Hazra, Ansuman Banerjee, S. M. C. R. M. (2008). Cohesive coverage management for simulation and formal property verification. *IEEE Computer Society Annual Symposium on VLSI*.

[5] Barbier, D. (2017). RISC-V Core IP Products. [online] https://www.sifive.com/core-designer.

[6] Brinkmann, R. (2012). How formal MDV can eliminate IP integration uncertainty. EE Times.

[7] Daniel D. Gajski, Samar Abdi, A. G. G. S. (2009). *Embedded System Design*. Springer.

[8] Drucker, L. (2002). Functional coverage metrics – the next frontier. EE Times.

[9] Group of Computer Architecture, U. o. B. (2018). RISC-V based Virtual Prototype. [online] https://github.com/agra-uni-bremen/riscv-vp.

[10] Hetalani, S. (2019). Correct-by-Construction Hardware Design of a Pipelined RISC-V CPU. Master's thesis, Technische Universität Kaiserslautern.

[11] Janick Bergeron, Eduard Cerny, A. H. and Nightingale, A. (2006). *Verification Methodology Manual for SystemVerilog*. Springer.

[12] Kaczynski, J. (2012). Don't be Afraid of UVM. [online] https://www.aldec.com/en/support/resources/multimedia/webinars/1502.

[13] Ludwig, T. (2018). SCAM. https://github.com/ludwig247/SCAM.

[14] Marcela Zachariasova, Lubos Moravec, J. S. and Jeeawoody, S. (2018). UVM-based Verification of a RISC-V Processor Core Using a Golden Predictor Model and a Configuration Layer. Mentor.

[15] OneSpin Solutions GmbH (2018). OneSpin 360 DV-Verify. [online] https://www.onespin.com/products/360-dv-verify/.

[16] Paolo Lenne, R. L. (2007). *Customizable Embedded Processors*. Morgan Kaufmann.

[17] Piziali, A. (2008). *Functional Verification Coverage Measurement and Analysis*. Springer.

[18] Stephan Schulz, Thilo Vörtler, M. B. (2015). UVM goes universal, introducing UVM in SystemC. [online] https://dvcon-europe.org/sites/dvcon-europe.org/files/archive/2015/proceedings/DVCon_Europe_2015_T05_Presentation.pdf.

[19] Tasiran, S. and Keutzer, K. (2001). Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45.

[20] Tobias Ludwig, Michael Schwarz, J. U. L. D. S. H. D. S. and Kunz, W. (2018). Property-Driven Development of a RISC-V CPU. *DVCON-EU*.

[21] YANG Yawen, ZHOU Shan, K. L. (2018). Coverage test technology based on onespin verification platform. *Journal of Physics*, 1026.

Appendix A

RISC-V Processor

A.1 Introduction

RISC-V is an open-source instruction set architecture (ISA) based on established reduced instruction set computing (RISC) principles. It's designed to be useful in a wide range of devices and has a substantial body of supporting software. The RISC-V ISA has been designed with small, fast, and low-power real-world implementations in mind, but without over-architecting for a particular microarchitecture style.

RISC-V instruction set is for practical computers, it has features to increase computer speed, yet reduce cost and power use. These include a load/store architecture, bit patterns to simplify the multiplexers in a CPU, simplified standards-based floating-point, a design that is architecturally neutral, and placing most-significant bits at a fixed location to speed sign-extension. The instruction set is designed for a wide range of uses. It supports three word-widths, 32, 64, and 128 bits, and a variety of subsets. Some of these subsets represent standard extensions defined to provide integer multiply/divide, atomic operations, and single and double-precision floating-point arithmetic. The base integer ISA is named "I" (prefixed by RV32 or RV64 depending on integer register width), and contains integer computational instructions, integer loads, integer stores, and control-flow instructions, and is mandatory for all RISC-V implementations. These subsets support small embedded systems, personal computers, supercomputers with vector processors, and warehouse-scale rack-mounted parallel computers.

The simplicity of the integer subset permits basic student exercises. The integer subset is a simple ISA enabling software to control research machines. The separated privileged instruction set permits research in operating system support, without redesigning compilers. In this research we will use the integer subset with 32-bit wide instructions (RV32I) with privileged instructions serving only machine mode to support interrupts handling and basic operating systems kernels like FreeRTOS [1]. In this chapter, we will focus on explaining the required additions to a previously built ISA in order to support interrupt handling.

A.2 Privileged Levels

At any time, a RISC-V hardware thread (*hart*) is running at some privilege level encoded as a mode in one or more CSRs [3]. Three RISC-V privilege levels are currently defined as shown in table A.1.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	*Reserved*	
3	11	Machine	M

Table A.1 RISC-V privilege levels.

Privilege levels are used to provide protection between different components of the software stack, and attempts to perform operations not permitted by the current privilege mode will cause an exception to be raised. These exceptions will normally cause traps into an underlying execution environment.

The machine level has the highest privileges and is the only mandatory privilege level for a RISC-V hardware platform. Code run in machine-mode (M-mode) is usually inherently trusted, as it has low-level access to the machine implementation. The simplest RISC-V implementations may provide only M-mode, though this will provide no protection against incorrect or malicious application code.

Many RISC-V implementations will also support at least user mode (U-mode) to protect the rest of the system from application code. Supervisor mode (S-mode) can be added to provide isolation between a supervisor-level operating system and the SEE.

A hart normally runs application code in U-mode until some trap (e.g., a supervisor call or a timer interrupt) forces a switch to a trap handler, which usually runs in a more privileged mode. The hart will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. In our design we will implement only M-mode, thus further considerations for U-mode and S-mode will not be mentioned.

A.3 Exceptions, Traps, and Interrupts

We use the term *exception* to refer to an unusual condition occurring at run time associated with an instruction in the current RISC-V thread (RISCV hardware thread means a single *hart* or core). We use the term *trap* to refer to the synchronous transfer of control to a trap handler caused by an exceptional condition occurring within a RISC-V thread. Trap handlers usually execute in a more privileged environment [2].

We use the term *interrupt* to refer to an external event that occurs asynchronously to the current RISC-V thread. When an interrupt that must be serviced occurs, some instruction is selected to receive an interrupt exception and subsequently experiences a trap. In our design we will focus on serving certain asynchronous exceptions (i.e. timer and external interrupts) only.

A.4 RV32I SYSTEM Instructions

SYSTEM instructions are used to access system functionality that might require privileged access and are encoded using the I-type instruction format. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other potentially privileged instructions.

A.4.1 CSR Instructions

The CSRRW (Atomic Read/Write CSR) instruction atomically swaps values in the CSRs and integer registers. CSRRW reads the old value of the CSR, then writes it to integer register *rd*. The initial value in *rs1* is written to the CSR. If *rd*=`x0`, then the instruction shall not read the CSR and shall not cause any of the side-effects that might occur on a CSR read.

The CSRRS (Atomic Read and Set Bits in CSR) instruction reads the value of the CSR, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected (though CSRs might have side effects when written).

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, and writes it to integer register *rd*. The initial value in integer register *rs1* is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in *rs1* will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable. Other bits in the CSR are unaffected.

The CSRRWI, CSRRSI, and CSRRCI variants are similar to CSRRW, CSRRS, and CSRRC respectively, except they update the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (uimm[4:0]) field encoded in the *rs1* field instead of a value from an integer register.

Some CSRs may be modified as side effects of instruction execution. In these cases, if a CSR access instruction reads a CSR, it reads the value prior to the execution of the instruction. If a CSR access instruction writes a CSR, the update occurs after the execution of the instruction.

A.4.2 Machine-Mode Privileged Instructions

Some of M-mode privileged instructions can be seen in table A.2, these instructions are encoded under the PRIV minor opcode.

31 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]		rs1	funct3	rd	opcode	I-type

Environment Call and Breakpoint

000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Trap-Return Instructions

0011000	00010	00000	000	00000	1110011	MRET

Table A.2 RISC-V Privileged Instructions

The ECALL instruction is used to make a request to the supporting execution environment, which is usually an operating system. The ABI for the system will define how parameters for the environment request are passed, but usually these will be in defined locations in the integer register file, a0, a1, a2, a7. In our design we try to build an implementation that can be synthesized into logic level, for this reason we designed an environment call handler that handles the basic required environment call like *exit* and *printf* [9]. The decision to design the environment call handler (referred to later as EcallHandler) in this manner comes from studying the required support for running FreeRTOS.

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. In our design we use this instruction to end simulations.

The MRET instruction is used to return after handling a trap. MRET sets the `pc` to the value stored in the `mepc` register.

A.5 Control Status Registers (CSR)

The standard RISC-V ISA sets aside a 12-bit encoding space (csr[11:0]) for up to 4,096 CSRs. Table A.3 list some of the CSRs that have currently been allocated CSR addresses and used in our design.

Number	Privilege	Name	Description
Machine Trap Setup			
`0x300`	MRW	`mstatus`	Machine status register.
`0x304`	MRW	`mie`	Machine interrupt-enable register.
`0x305`	MRW	`mtvec`	Machine trap-handler base address.
Machine Trap Handling			
`0x341`	MRW	`mepc`	Machine exception program counter.
`0x342`	MRW	`mcause`	Machine trap cause.
`0x344`	MRW	`mip`	Machine interrupt pending.

Table A.3 Some of allocated RISC-V machine-level CSR addresses.

Machine Status Register (`mstatus`)

The `mstatus` register is an 32-bit read/write register formatted as shown in figure A.1. The `mstatus` register keeps track of and controls the hart's current operating state. Interrupt-enable bit, MIE, is used to guarantee atomicity with respect to interrupt handler. The MRET instruction is used to return from traps.

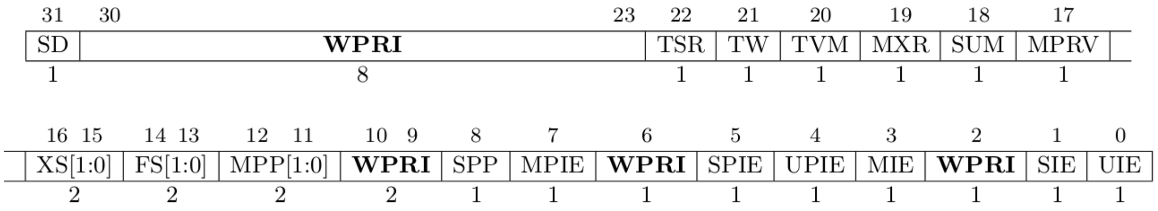

31 30 23 22 21 20 19 18 17
SD | WPRI | TSR | TW | TVM | MXR | SUM | MPRV
1 8 1 1 1 1 1 1
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
XS[1:0] | FS[1:0] | MPP[1:0] | WPRI | SPP | MPIE | WPRI | SPIE | UPIE | MIE | WPRI | SIE | UIE
2 2 2 2 1 1 1 1 1 1 1 1 1

Fig. A.1 Machine-mode status register (`mstatus`) for RV32.

Machine Trap-Vector Base-Address Register (`mtvec`)

The `mtvec` register is an XLEN-bit read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE). The `mtvec` register must always be implemented, but can contain a hardwired read-only value. If `mtvec` is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field.

XLEN-1 2	1 0
BASE[XLEN-1:2] (**WARL**)	MODE (**WARL**)
XLEN-2	2

Fig. A.2 Machine trap-vector base-address register (`mtvec`).

Value	Name	Description
0	Direct	All exceptions set pc to BASE.
1	Vectored	Asynchronous interrupts set pc to BASE+4×cause.
≥2	—	*Reserved*

Table A.4 Encoding of `mtvec` MODE field.

The encoding of the MODE field is shown in Table A.4. When MODE=Direct, all traps into m-mode cause the `pc` to be set to the address in the BASE field. When MODE=Vectored, all synchronous exceptions into machine mode cause the `pc` to be set to the address in the BASE field, whereas interrupts cause the `pc` to be set to the address in the BASE field plus four times the interrupt cause number.

Machine Interrupt Registers (`mip` and `mie`)

The `mip` register is an XLEN-bit read/write register containing information on pending interrupts, while `mie` is the corresponding XLEN-bit read/write register containing interrupt enable bits.

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	**WIRI**	SEIP	UEIP	MTIP	**WIRI**	STIP	UTIP	MSIP	**WIRI**	SSIP	USIP
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Fig. A.3 Machine interrupt-pending register (`mip`).

The MTIP bit correspond to timer interrupt-pending bit for machine timer interrupt. The MTIP bit is read-only and is cleared by writing to the memory-mapped machine-mode timer compare register. The MEIP field in `mip` is a read-only bit that indicates a machine-mode external interrupt is pending. MEIP is set and cleared by platform-level interrupt controller (PLIC). An interrupt *i* will be taken if bit *i* is set in both `mip` and `mie`, and if interrupts are globally enabled (i.e. MEI is set in `mstatus`).

Multiple simultaneous interrupts and traps at the same privilege level are handled in the following decreasing priority order: external interrupts, software interrupts, timer interrupts, then finally any synchronous traps.

Machine Timer Registers (`mtime` and `mtimecmp`)

Platforms provide a real-time counter, exposed as a memory-mapped machine-mode register, `mtime`. `mtime` must run at constant frequency, and the platform must provide a mechanism for determining the timebase of `mtime`.

The `mtime` register has a 64-bit precision. Platforms provide a 64-bit memory-mapped M-mode timer compare register (`mtimecmp`), which causes a timer interrupt to be posted when the `mtime` register contains a value greater than or equal to the value in the `mtimecmp` register. The interrupt remains posted until it is cleared by writing the `mtimecmp` register. The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the `mie` register.

Machine Exception Program Counter (`mepc`)

`mepc` is an XLEN-bit read/write register. When a trap is taken into M-mode, `mepc` is written with the virtual address of the instruction that encountered the exception. Otherwise, `mepc` is never written by the implementation, though it may be explicitly written by software.

Machine Cause Register (`mcause`)

The `mcause` register is an XLEN-bit read-write register formatted as shown in figure A.4. When a trap is taken into M-mode, `mcause` is written with a code indicating the event that caused the trap. Otherwise, `mcause` is never written by the implementation, though it may be explicitly written by software.

XLEN-1	XLEN-2 0
Interrupt	Exception Code (**WLRL**)
1	XLEN-1

Fig. A.4 Machine Cause register `mcause`.

The Interrupt bit in the `mcause` register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception. Table A.5 lists some of the possible machine-level exception codes.

A.6 Platform-Level Interrupt Controller (PLIC)

Figure A.5 provides a quick overview of PLIC operation, the figure shows the first two of potentially many interrupt sources, and the first of potentially many interrupt targets. The

Interrupt	Exception Code	Description
1	3	Machine software interrupt.
1	7	Machine timer interrupt.
1	11	Machine external interrupt.
1	≥12	*Reserved*

Table A.5 Machine cause register (`mcause`) values after interrupt.

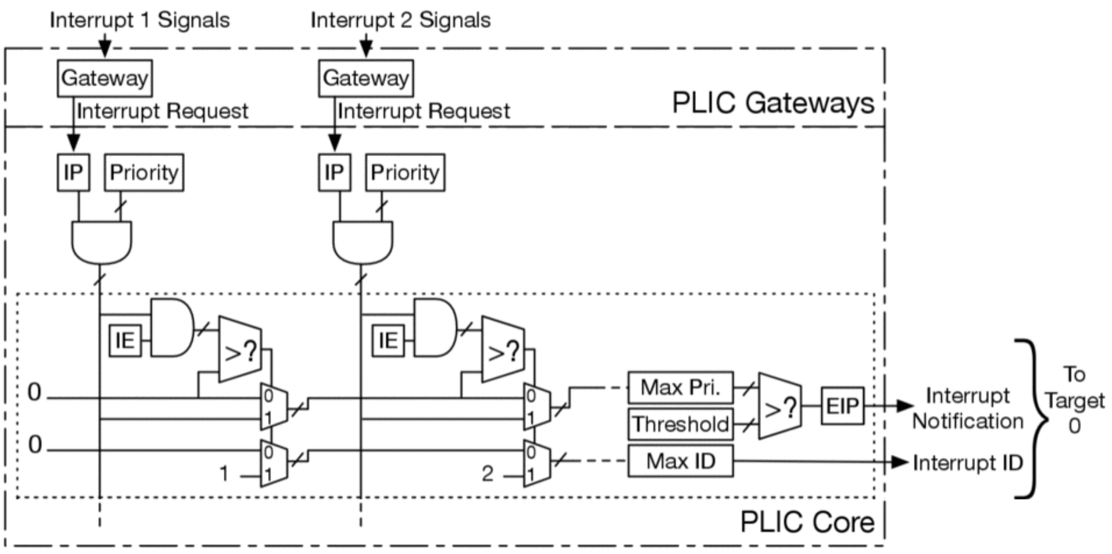


Fig. A.5 Platform-Level Interrupt Controller (PLIC) conceptual block diagram.

PLIC connects global *interrupt sources*, which are usually I/O devices, to *interrupt targets*, which are usually *hart contexts*. The PLIC contains multiple *interrupt gateways*, one per interrupt source, together with a PLIC *ore* that performs interrupt prioritization and routing. Global interrupts are sent from their sources to their corresponding *interrupt gateways*, each of which processes their interrupt signals and sends a single *interrupt request* to the PLIC core, which latches these in the core interrupt pending bits (IP). Each interrupt source is assigned a separate priority. The PLIC core contains a matrix of interrupt enable (IE) bits to select the interrupts that are enabled for each target. The PLIC core forwards an *interrupt notification* to one or more targets if the targets have any pending interrupts enabled, and the priority of the pending interrupts exceeds a per-target threshold. When the target takes the external interrupt, it sends an *interrupt claim* request to retrieve the identifier of the highest-priority global interrupt source pending for that target from the PLIC core, which then clears the corresponding interrupt source pending bit. After the target has serviced the interrupt, it sends the associated interrupt gateway an *interrupt completion message* and the interrupt gateway can now forward another interrupt request for the same source to the PLIC.

A.7 RISCV Complex System-Level Model

Following the rules of SystemC-PPA and in compliance with SiFive [5] guidelines regarding the structure and memory mapping of RISCV processor, we created our complex system-level design for a RISCV processor with interrupts handling, fig. A.6. We will briefly describe the modules and sub-modules created for this design and the communication channels used between them.

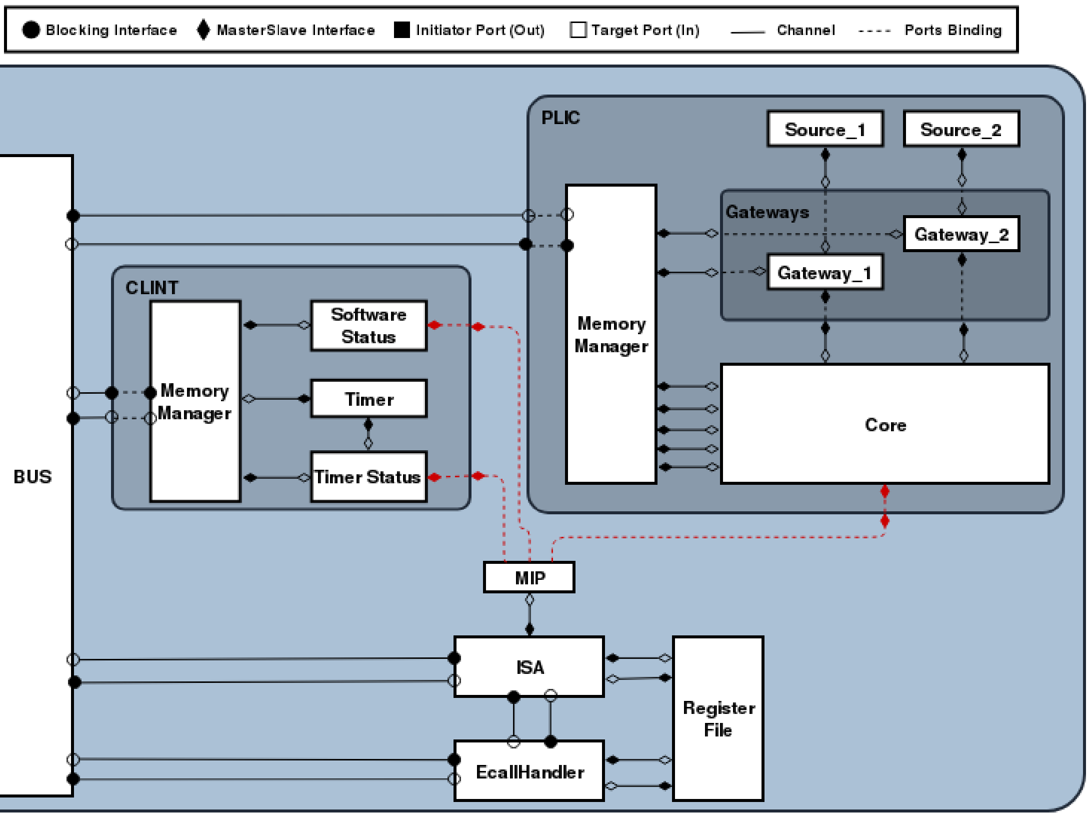


Fig. A.6 RISCV core design following SystemC-PPA.

ISA

The main module of the core, it handles all the control and arithmetic executions in the core. It communicates with other modules through the following ports:

- *blocking_out <CUtoME_IF> toMemoryPort* used for request a read instruction, request a read data for load instructions and request a write data for store instructions.
- *blocking_in<MEtoCU_IF> fromMemoryPort* used for read instruction and read data for load instructions.

R_Type	RF_read	···	···	RF_write	···	MIP_read
B_Type	RF_read	···	···	···	···	MIP_read
S_Type	RF_read	···	**MEM_write**	···	···	MIP_read
U_Type	···	···	···	RF_write	···	MIP_read
J_Type	···	···	···	RF_write	···	MIP_read
I_I_Type	RF_read	···	···	RF_write	···	MIP_read
I_L_Type	RF_read	···	**MEM_read**	RF_write	···	MIP_read
I_J_Type	RF_read	···	···	···	···	MIP_read
I_S_Type 1	RF_read	···	···	RF_write	···	MIP_read
I_S_Type 2	···	···	···	RF_write	···	MIP_read
I_S_Type 3 (ECALL)	···	···	···	···	**EcallHandler**	MIP_read
I_S_Type 3	···	···	···	···	···	MIP_read

Table A.6 ISA communications stages

- *master_in<RegfileType> fromRegsPort* used for reading the register file.
- *master_out<RegfileWriteType> toRegsPort* used for writing back to the register file.
- *blocking_out<bool> isa_ecall_Port* used for triggering the EcallHandler when an ECALL is read. This communication has to be through a Blocking channel in order to suspend execution of ISA while the ECALL is being served.
- *blocking_in<bool> ecall_isa_Port* used for reading the status of the processor after ECALL. This status might trigger the end of simulation for system-level model when the environment call served is *exit*.
- *master_in<unsigned int> mip_isa_Port* used for reading the status of the connected interrupt sources.

The sequence of communications with other modules differ depending on the instruction being executed. These instructions can be categorized according to their encryption type. Table A.6 shows the sequence of communications for all possible encryption types following the pair of communication to read an instruction from memory.

Register file

This module will always present the content of the register file to the connected modules (ISA and EcallHandler) by using *slave_out* ports. It will also keep on checking both inward ports for writing requests. It has the following communication ports:

- *slave_in<RegfileWriteType> toRegsPort* used for reading a write request from ISA to a certain register in the register file.

- *slave_out<RegfileType> fromRegsPort* used for exporting the entire register file to ISA.
- *slave_out<RegfileEcallType> reg_ecall_Port* used for exporting registers (`x10`, `x11`, `x12` and `x17`) to EcallHandler.
- *slave_in<RegfileWriteType> ecall_reg_Port* used for reading a write request from EcallHandler to register (`x10`) in the register file.

Environment call handler

This module is responsible for executing environment requests from the running program. These request may include many types, but in our current implementation we are only supporting both (*printf* and *exit*). It has the following communication ports:

- *blocking_out<CUtoME_IF> toMemoryPort* used for requesting a read of memory content.
- *blocking_in<MEtoCU_IF> fromMemoryPort* used for reading data from the memory.
- *master_in<RegfileEcallType> reg_ecall_Port* used for reading registers (*x10*, *x11*, *x12* and *x17*) which should be holding the arguments and type of ecall request.
- *master_out<RegfileWriteType> ecall_reg_Port* used for writing back to the register file (specifically to `x10`, which should hold the result of an ecall request).
- *blocking_out<bool> ecall_isa_Port* used for returning to ISA the status of the processor after handling the ecall.
- *blocking_in<bool> isa_ecall_Port* used for triggering the main process of the EcallHandler by ISA.

Pending machine interrupt handler

This module is responsible for collecting the interrupts statuses (In our current implementation we are only supporting system, timer and external interrupts), and present them to ISA as a single 32-bit value (`mip` register of CSRs) with each bit referring to a specific interrupt source. It has the following communication ports:

- *slave_out<unsigned int> mip_isa_Port* used for writing the 32-bit value to ISA.
- *master_in<bool> MSIP_port* used for reading the status of system interrupt.

- *master_in<bool> MTIP_port* used for reading the status of timer interrupt.
- *master_in<bool> MEIP_port* used for reading the status of external interrupt.

Core Local Interruptor (CLINT)

This module is used to generate software and timer Interrupts. It contains the RISC-V `msip`, `mtime` and `mtimecmp` memory mapped CSRs. The `msip` memory mapped CSR can be used to generate Machine Software Interrupts (MSIP). This register can be accessed by remote harts to provide machine-mode interprocessor interrupt (Not included in our current implementation). `mtime` and `mtimecmp` memory mapped CSRs can be used to generate Machine Timer Interrupts (MTIP). This interrupt is set when `mtime` exceed `mtimecmp` and it can be reset by writing a new bigger value to `mtimecmp`.

This module has the following communication ports:

- *blocking_in<CUtoME_IF> COtoME_port* used for reading memory access requests from the Core.
- *blocking_out<MEtoCU_IF> MEtoCO_port* used for presenting the CLINT memory reply to the Core.
- *slave_out<bool> MSIP_port* used for presenting the MSIP status to the Core.
- *slave_out<bool> MTIP_port* used for presenting the MTIP status to the Core.

CLINT requires more than one thread to be running simultaneously and because currently SystemC-PPA doesn't support multiple threads, the internal of CLINT has been divided into submodules as shown in figure A.6.

Platform Level Interrupt Controller (PLIC)

This module is used to prioritize and distribute global interrupts and generate on their behalf what is called the External Interrupt (MEIP). For every added external interrupt source, a new gateway should be added to the *PLIC_Gateways* to handle the interrupt type (i.e. Level triggered or edge triggered interrupts) and required register and channels to *PLIC_Core* and *PLIC_Memory_Manager* follows as well. This module has the following communication ports:

- *blocking_in<CUtoME_IF> COtoME_port* used for reading memory access requests from the Core.

- *blocking_out<MEtoCU_IF> MEtoCO_port* used for presenting the PLIC memory reply to the Core.
- *slave_out<bool> MEIP_port* used for presenting the MEIP status to the Core.

PLIC require more than one thread to be running simultaneously and because currently SystemC-PPA doesn't support multiple threads, the internal of PLIC have been divided into submodules as shown in figure A.6.

CoreBus

This module is responsible for forwarding ISA and EcallHandler memory access requests and replies to the three connected memory mapped modules (CLINT, PLIC and Memory). It has the following communication ports:

- *blocking_in<CUtoME_IF> ecall_bus_Port* used for reading the EcallHandler memory access requests.
- *blocking_out<MEtoCU_IF> bus_ecall_Port* used for forwarding the memory reply to the EcallHandler.
- *blocking_in<CUtoME_IF> isa_bus_Port* used for reading the ISA memory access requests.
- *blocking_out<MEtoCU_IF> bus_isa_Port* used for forwarding the memory reply to the ISA.
- *blocking_out<CUtoME_IF> BUStoMEM_port* used for forwarding the Core memory access requests to Memory.
- *blocking_in<MEtoCU_IF> MEMtoBUS_port* used for reading the Memory reply.
- *blocking_out<CUtoME_IF> BUStoCLINT_port* used for forwarding the Core memory access requests to CLINT.
- *blocking_in<MEtoCU_IF> CLINTtoBUS_port* used for reading the CLINT reply.
- *blocking_out<CUtoME_IF> BUStoPLIC_port* used for forwarding the Core memory access requests to PLIC.
- *blocking_in<MEtoCU_IF> PLICtoBUS_port* used for reading the PLIC reply.

Adding new Peripherals will require adding two new communication ports to this module and manage the linking at the Core level.

A.8 RTL implementation and properties refinements

Most of the ESL modules of the RISCV-Interrupt processor have been designed following SystemC-PPA methodology with the exception of some internal submodules in CLINT and PLIC which are considered as environmental components and not as parts of the main core or the RISCV processor. Starting from these ESL modules, a complete set of ITL properties have been generated for each sub-module using *SCAM* tool and later the RTL design of these sub-modules have been created while guaranteeing that the properties are holding. The only refinement needed of the properties was to modify the macros to match the RTL naming for ports and signals. No timing refinement was needed because this RISCV processor implementation was designed with single cycle processes. The tool used for verifying the ITL properties was OneSpin [15].