Functional Verification of a MIPS CPU using UVM

EGC447 Functional Verification of Hardware Systems

George Dagis (CE), NAME OMITTED (CE)

May 17th, 2019

Instructor: Yi-Chung Chen

Abstract

This project outlines the simulation and functional verification of a MIPS processor by using universal verification methodology (UVM). The MIPS processor discussed within the project has been designed with system verilog and has been previously simulated using Questa through a Linux virtual machine using a custom testbench. In this project a custom UVM Development Environment has been designed in order to test multiple instructions which are made possible by the simulated processor.

Table of Contents

1. Introduction

Universal Verification Methodology (UVM) is a methodology for functional verification using SystemVerilog classes. It supports constrained random verification. The Universal Verification Methodology uses a set of transaction-level communication interfaces. This allows for the connection of components at the transaction level. This verification process is unique and useful because it serves as a passive debugging tool for issues which were not looked for. It enables users to force their design into states that were not previously thought of. UVM requires a knowledge of Object Oriented Programming (OOP) and Transaction Level Modeling (TLM) for most efficient use.

1.1 Object Oriented Programming

Object Oriented Programming is a programming approach which suggests the user base their design off of "objects" in order to make code less abstract and easier to work with. The OOP approach is based around four principles: abstraction, encapsulation, inheritance, and polymorphism. These principles serve the purpose of relating the objects together within a system to make complex systems more manageable. For example, inheritance suggests one set of objects may be treated like another separate set of objects "except for a few differences." Inheritance speeds up the development process by encouraging code reuse.

1.2 Transaction Level Modeling

Transaction Level Modeling is a high-level approach used to model digital systems. This process hides the details of how the processor actually works (clock signals and data pins) and

instead transfers register data. TLM simulates quickly because of its level of abstraction. It gives the user the ability to model transactions in digital systems based on a memory mapped bus network.

2. Design

A 32-Bit MIPS processor is the design under test (DUT) and is used for verification. UVM is to be used to verify the processor. The general design of this system follows the idea illustrated from Figure 1, below.

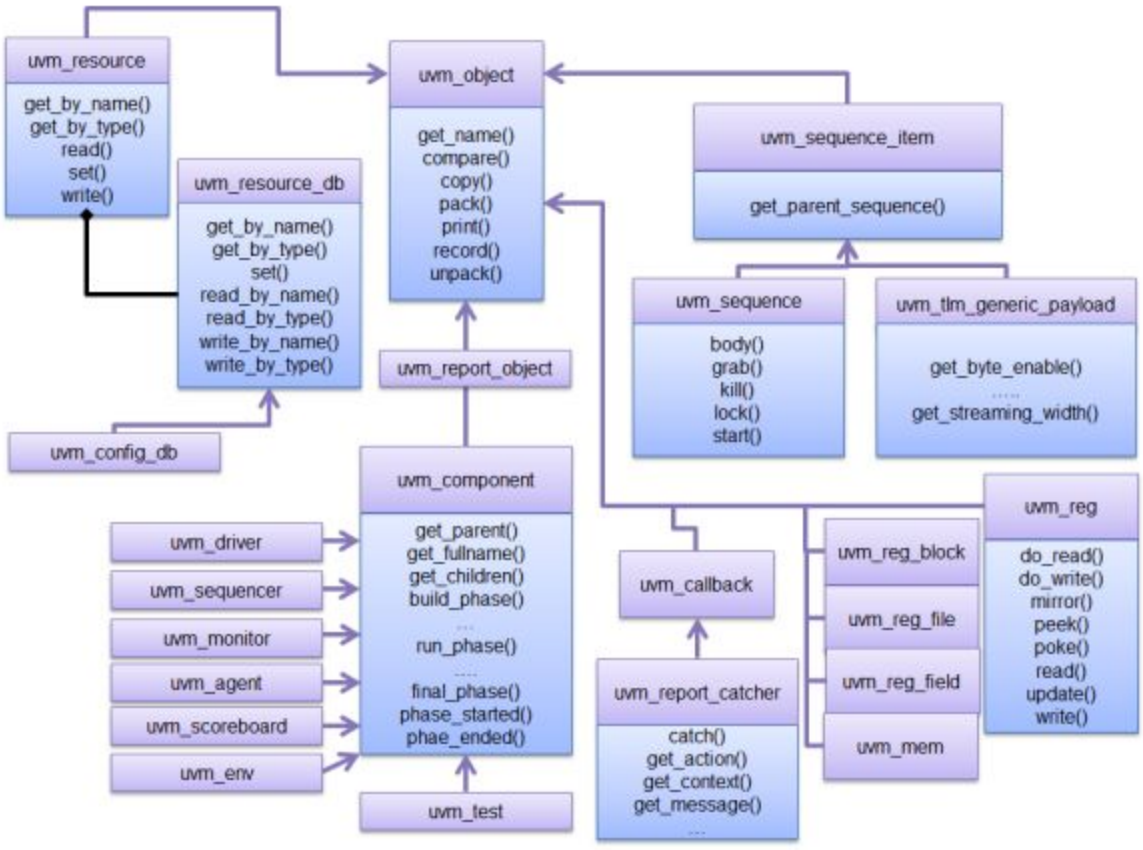

Figure 1: UVM Hierarchy

The UVM is a refined engine with many working modules. Each module is essential to ensure that the processor is working properly. The top.sv, seq_item.sv and sequence.sv SystemVerilog files are examples of the most foundational and important modules in this case and will be further explained in sections 2.1 - 2.3, below.

2.1 top.sv

The top.sv functions as the wrapper for the UVM environment. Here, everything is instantiated, clock and reset are set, and the virtual interface is set up.

Another very important part of the code is contained in top.sv. That is the UVM specific command: run_test. This test starts the simulation. This is better illustrated in Figure 2, below.

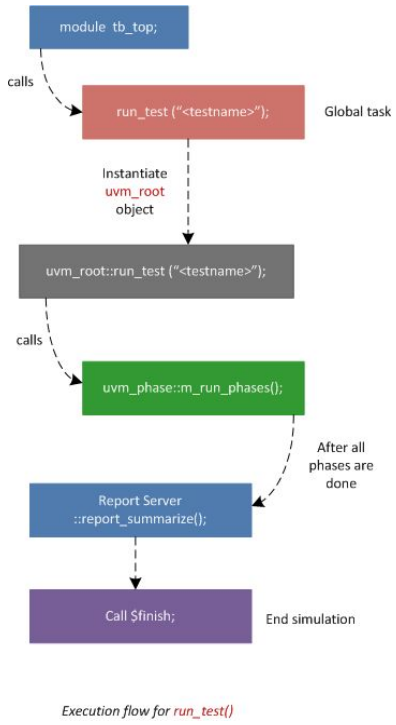

Figure 2: run_test Logic

2.2 seq_item.sv

The UVM class seq_item is the base class for the UVM sequence class. This class provides the essential functionality for objects and other elements to operate in the sequence mechanism.

This part of the code is where most of the constraint driven random logic is originated. Making use of the verilog command constraint, one can test the received opcode, and generate a random instruction base on that opcode.

The code also has a convert2string section. Here, the instruction funct section is read, and the appropriate instruction name is output. This can be seen in Figure 3, below.

```
function string convert2string;
  case(instr[31:26])
    6'd8  : instr_name = "ADDI";
    6'd9  : instr_name = "ADDIU";
    6'd10 : instr_name = "SLTI";
    6'd11 : instr_name = "SLTIU";
    6'd12 : instr_name = "ANDI";
    6'd13 : instr_name = "ORI";
    6'd14 : instr_name = "XORI";
    6'd32 : instr_name = "LB";
    6'd33 : instr_name = "LH";
    6'd34 : instr_name = "LWL";
    6'd35 : instr_name = "LW";
```

Figure 3: convert2string code in seq_item

2.3 sequence

The UVM class sequence is responsible for generating sequence_item's and sending them to the driver. This is all done using the class sequencer. In Figure 4 on the next page, one can see the handshake process between the sequencer and the driver.

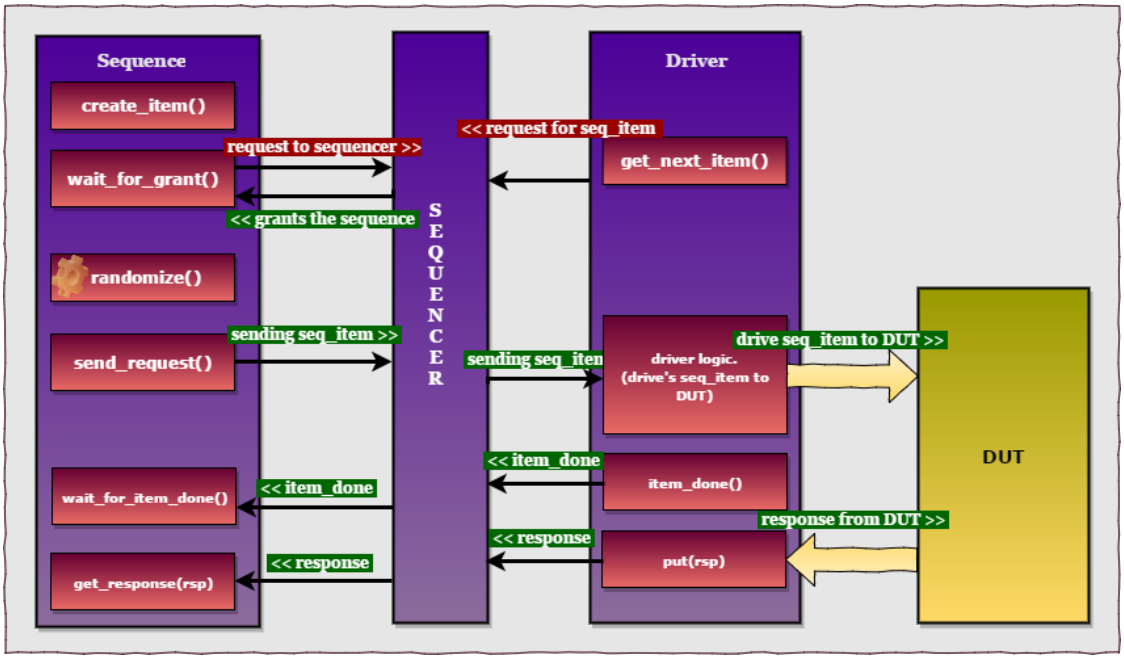

Figure 4: Handshake Between the Sequencer and the Driver.

The process can be broken down into the following steps show in Table 1.

Table 1: Handshake process broken down.

1	create_item() / create request
2	Wait for grant
3	Randomize the request
4	Send the request
5	Wait for item
6	Get response

3. Results

The output of the group's UVM program lists whether or not and instruction is read and the processor is actually receiving the code it's expected to. In Figure 5, below, the result of the UVM is shown.

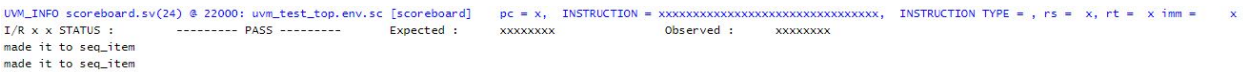

Figure 5: Result of UVM

In an ideal system, the processor would recognize and lead to displaying a known, definitive instruction and compare it to another known, definitive instruction from the UVM. In the system outlined here the instruction is not properly recognized. The group believes this is because of the hand-shaking process mentioned earlier in the report. This is thought to be the case because the group had discussed with other groups which did not have the issue and thought it could be the cause.

4. Discussion

While the design mostly works as desired, some things can be learned from the design process Some reflections can be made about our design and many things were reinforced throughout designing the project.

4.1 Problems Encountered

Because this was the first time working with UVM, the group's original plan was to study a working example of a UVM environment and use that as a reference as the group's model. In our research we came across RISC-V processor with a UVM environment made specifically for

it. The group then had the UVM randomly test a few simple instructions and increase the amount of instructions after successfully testing the ones before.

The biggest problem encountered was the handshaking between the sequencer and driver. This handshake signal must be completed before another item is taken from the sequencer. It wasn't discovered until late in the group's testing that the code could work without this mechanism. Unfortunately the handshake was too deeply interwoven in the code to remove it completely. Given more time the group could have overcome this or gotten the handshake to function properly. Figure 6, below, shows one of the handshake sections of our code in the driver class.

```
task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    drive(req);
    seq_item_port.item_done();
  end
endtask
```

Figure 6: Handshake Request in Driver

As stated in the results section, future improvements could be to resolve the 'x' don't care values by removing the handshaking process.

4.2 Lessons Learned

This project taught the group lot about UVM, OOP, TLM, and functional verification itself. Another skill improved through this project was SystemVerilog coding. Most of the group's struggles were in the research phase of the project.

The group learned that doing proper research ahead of time in order to overcome coding issues later on in the development process is necessary.

Bibliography

[1] Yi-Chung Chen's Website. Functional Verification of Hardware Notes. [online] Available at: http://engr.newpaltz.edu/~cheny/EGC4472019S/note447.html [Last Accessed, 15 May, 2019]

[2] Aldec. Recorded Webinars. [online] Available at: https://www.aldec.com/en/support/resources/multimedia/webinars [Last Accessed, 15 May, 2019]

[3] Grantae's Github Page. mips32r1_core. [online] Available at: https://github.com/grantae/mips32r1_core [Last Accessed, 15 May, 2019]

[4] Verification Guide. UVM Sequencer. [online] Available at: www.verificationguide.com/p/uvm-sequencer.html [Last Accessed, 15 May, 2019]

[5] Chip Verify. How run_test( ) Starts the Simulation ? - Blog." ChipVerify, [online] Available at: www.chipverify.com/blog/how-run-test-starts-the-simulation-1 [Last Accessed, 15 May, 2019]

[6] Tala, Deepak Kumar. Transaction Level Modeling Part I, 1 Feb. 1970, [online] Available at: www.asic-world.com/systemc/tlm1.html [Last Accessed, 15 May, 2019]

[7] Tech Design Forum Techniques. Transaction Level Modeling. [online] Available at: www.techdesignforums.com/practice/guides/transaction-level-modelling-tlm/ [Last Accessed, 15 May, 2019]

[8] Universal Verification Methodology (UVM) 1.2 User's Guide. [online] Available at: https://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf [Last Accessed, 15 May, 2019]

[9] Verification Academy. uvm_sequence_item. [online] Available at: verificationacademy.com/verification-methodology-reference/uvm/docs_1.1a/html/files/seq/uvm_sequence_item-svh.html [Last Accessed, 15 May, 2019]