RESEARCH ARTICLE OPEN ACCESS

Cost Effective Design & Verification of Ram Using Eda Simulator Icarus with Cocotb

K. Srinath [1], Dr. R. Anandan [2], Vijitha S [3]. R. Deepa [4]

.[1] M.E CSE First Year, VISTAS, Chennai
[2] Professor-CSE,VISTAS, Chennai
[3] Assistant Professor-CSE,VISTAS,Chennai
[4] Assistant Professor-CSE, VISTAS, Chennai

ABSTRACT

A CPU, RAM, ROM, and other componentsare frequently included in electronic circuits on a single PCBA. Yet, an integrated circuit designer has the option to combine all ofthese into a single chip thanks to very large- scale integration (VLSI) technology. To model electronic systems and validate digital circuitry at the register-transfer level of abstraction, one uses the HDL Verilog. Analog and mixed signal circuits can bothbe verified using it. Nowadays, computers are widely used in all fields of design. Structures are so complicated that physical labor is no longer even an option in our minds. This is the current trend throughout all engineering disciplines, not justelectronic ones. The time when circuitdesigners could test their designs on a breadboard is long gone.

The Verilog hardware descriptionlanguage compiler is implemented in Icarus Verilog, which produces netlists in therequired format (EDIF). It supports the 1995, 2001, and 2005 revisions of the standard, as well as various extensions and a few system Verilog modules. The main benefit of Icarus is that there are no costs associated with access or license. Icarus simulator can be used to create Memory.The RAM design must then be confirmed in order to evaluate RAM performance using cocotb. Productivity in verification is the focus of the cocotb. Because verification is a form of software, verification engineers have access to all the Python deliciousness that has madesoftware development efficient. It enablesdevelopers to quit battling language restrictions. Python is used to create RAM test cases and verification code with cocotb. Cocotb offers a lean framework to effectively create verification code in addition to all the benefits of the Python programming language and its ecosystem. Icarus has a support of cocotb verification, it can easily make a communication with cocotb. So we can achieve cost effective design &verification of RAM.

OBJECTIVE OF THE PROPOSED SYSTEM

The primary goal of this proposed system is to produce RAM design and verification at no expense. There are numerous software tools for circuit design accessible on the market, but their licenses come at a hefty price. In the field of verification, businessescurrently employ UVM (Universal Verification Methodology) software, which is difficult to learn and more difficult to operate. Hence, cocotb uses Python as a verification language and replaces UVM inthe verification of RAM circuit design.

Icarus enables Verilog RAM design creation and, more importantly, it supports cocotb. The permissive BSD license applies to cocotb, which is open source. It is open source and free to use, modify, and distribute for any reason, including closed- source and commercial applications.

- ✓ The objective of the proposed system is to create cost-effective RAM.
- ✓ To make the electronic design automation industry even smarterand easier.
- ✓ To reduce the verification coding pages & any one can learn easily.

MOTIVATION

Most business owners lack the capital required to start a venture in the technology industry. They will worry about needing to spend more money in order to get the license for the software tool they require. They have the skills needed to be successful entrepreneurs, but they give up on their dream due to financial constraints. They will use some illegal software to reduce costs, but it won't give them full access. I therefore worked on the suggested system for thosereasons.

INTRODUCTION

Today, computers are an integral part of every design discipline. This is the current trend across all engineering disciplines, not just electronics. Designers saw the need for a language that could represent digital circuits in the 1980s so that the circuit could be analyzed and simulated. Hardware description languages (HDL) were created in this manner. I'll introduce you to an open- source Verilog simulation and synthesizer tool in this article. HDLs by themselves were not enough to lighten the load on VLSI circuit designers. Quickly, tools for logic synthesis were created that could convert an HDL-based design into a schematic circuit.

The fact that hardware description languages provide techniques to describe how time and signal dependencies spread sets them apart from software programming languages. Using HDL, designs are created at a very abstract level and are not dependent on the IC fabrication method. It was initially designed as a simulation language, and synthesis support was only later added. Very High-Speed Integrated Circuit Hardware Descriptive Language, or VHDL for short, is a programming language used to describe the structure, behavior, and functions of a logic circuit. An implementation of the Verilog hardware description language is called Icarus Verilog or Verilog.

There are versions of Icarus Verilog for Linux, Windows, and Mac OS. Seeing the simulated output of the Verilog code is done using GTK Wave, a fully featured waveform viewer built on the GTK+ framework. Icarus The source RAM design is written in Verilog, and Verilog is a command-line tool that compiles it to the target format. Icarus is completely free to access so anyone can create their own circuit design without any cost.

Then verification of RAM design can be achieved through cocotb. The open-source project cocotb was created by a loosely knit group of technologists with similar viewpoints. value bug reports, feature requests, customer support, and documentation updates just as much as we value code changes or assistance with releasing and maintaining cocotb. cocotb can simulate any (RAM) hardware design, whether it is created in (System)Verilog, VHDL, a mixed language, or even a mixed- signal design. cocotb supports all main simulators. RAM Python test cases and proof code are written using cocotb. Along with all the benefits of the Python computer language and its ecosystem, cocotb offers a simple framework for quickly writing verification code. The built-in test runner with CI integration in cocotb makes running RAM tests simpler than ever. As an alternative, it's simple to integrate cocotb with your current build and reporting system. We develop this Concept using EDA playground for the Effective Results.

SYSTEM ANALYSIS

EXISTING SYSTEM

Nowadays, VLSI industries employ cadence software. Even though it was a strong tool, the cost of the license was higher. The largest percentage of the total expenditure is spent on engineering design and verification for RAM. The nature of RAM hardware design necessitates the creation of several dynamic parallel processes, which are common in UVM testbench results. As most of these processes are relatively little in size, it wouldn't be beneficial to try to generate separate CPU threads for each one and synchronize them.

It is quite challenging to verify Memory using the System Verilog language. Almost a thousand pages make up the System Verilog specification. Compared to C++'s
83 keywords, language includes 221 keywords. Strong but difficult to learn.

DISADVANTAGE:

- The expense of the licensing is the biggest disadvantage of utilizing Cadence Xelium for Memory design.
- The libraries used by this program are quite difficult to use and require that we keep thousands of data.
- Because UVM components are more numerous and the verification procedure is time-consuming, more RAM code is needed.

PROPOSED SYSTEM:

In our proposed system, To get around these difficulties, we're suggesting two pieces of software: ICARUS for designing RAM and cocotb for verifying RAM circuits. In essence, this will establish and keep a connection with cocotb. ICARUS in the EDA playground permits cocotb for the RAM design verification methodology. For the design and verification of hardware circuits, businesses all spend money on licensed software. Even students require some illegal software for their academic needs. cocotb removes the python language in favor of the system Verilog, allowing for more code to be written in the verification area with less time spent on it. No additionalRTL code is necessary for a typical Cocotb testbench.

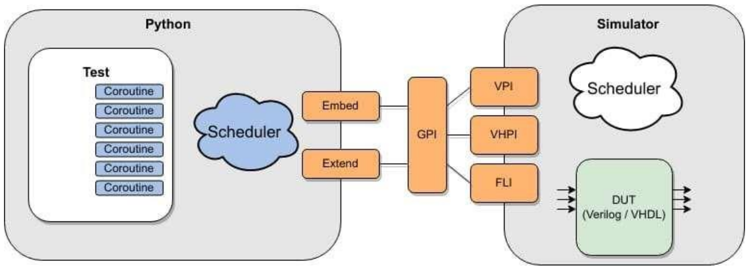
The Design Under Test (DUT) isinstantiated as the top-level of the simulatorwithout the need of any wrapper code. The DUT's inputs (or those lower in the hierarchy) are stimulated by Cocotb, which uses Python to directly track the outputs. In order to connect Python and the simulator, cocotb is used. It uses the VHDL Procedural Interface (VHDLPI) or the Verilog Procedural Interface (VPI) (VHPI). Simply put, a test is a Python function.

The await keyword specifies when the simulator should regain control of execution. Several coroutines can be started by a test, allowing for multiple execution flows.

ADVANTAGE:

- Users can modify, simulate (and examine waveforms), synthesize, and share their HDL code using the free web tool ICARUS EDA Playground.
- Updating values when moving through the DUT hierarchy. The simulationtimer will eventually expire. Observe the rising or lowering edge of the signal.
- It automatically finds tests; thus, an extra step is not needed to include a test in a regression.
- When compared to UVM, it requires less skill and reduces coding complexity.

Proposed system Architecture:



MODULES:

- RTL module code (Verilog).
- Cocotb Coroutine module(python).
- Top rapper module (Verilog**).**

Modules description:

RTL MODULE CODE(VERILOG):

Using clock-based write and read operations, this module will generate the RTL RAM design in Verilog. Details and specifications for the RAM design are contained in this module. It determines the RAM's capacity and functionality. It offers edges, a reset, an address, and the size of read and write data. If customers want to add any other functionality, they can do so in this module. Normally, the working functionality of RAM is excluded from the design phase.

COCOTB COROUTINE MODULE(PYTHON):

Python-coded RAM verification is contained in this module. The randomized variable is passed during read and write operations for verification, and this module then returns the variable about the RAM's verification code**.** Cocotb enables you to directly control the signals in your design from Python by automatically establishing connections to a range of HDL simulators (including Icarus). Python may be used to write the entire testbench, and the ease of implementing automation and randomization will enhance productivity.

It is an independent simulation for the test bench and design. For communication, **VPI/VHPI** interfaces—represented as Cocotb "triggers"—are employed. The simulation time does not advance while the Python function is running. The testbench pauses execution after a trigger is provided until the triggered condition is met before restarting it. Since the hierarchy can be easily ascended because Python and RTL are co-simulated. Any internal signal can be read or altered by the Python testbench. It facilitates single event upset modelling. With Cocotb, post-synthesis simulations are also possible. With the wrapper technique, time constraints (SDF) files can be loaded on demand.

TOP RAPPER MODULE:

This module will connect the RAM DUT with Cocotb RAM verification code (design under test). RTL testbench components may still be utilized. The actual design being tested must be instantiated along with additional testing components and a trigger interface in order to produce a top-level Verilog or VHDL logic. Although it is not feasible to call operations directly, it can still be useful for low-level testing, assertions, and other purposes. With Cocotb, post-synthesis simulations are also possible. The wrapper method enables on-demand loading of time constraints (SDF) files.

Hardware Environment:

Since they could serve as the foundation of a contract for the system's implementation, the hardware requirements ought to be a thorough and consistent description of the entire system. For software engineers, they serve as the basis for system design. It does not explain how the systems should be used; rather, it shows what the systems do**.**

- **CPU | Any x86 processor**
- **Architecture | 64 bit**
- **Base Clock Speed | 1.8 GHz**
- **HDD or SSD space | 32 Gb**
- **RAM | 4 Gb**

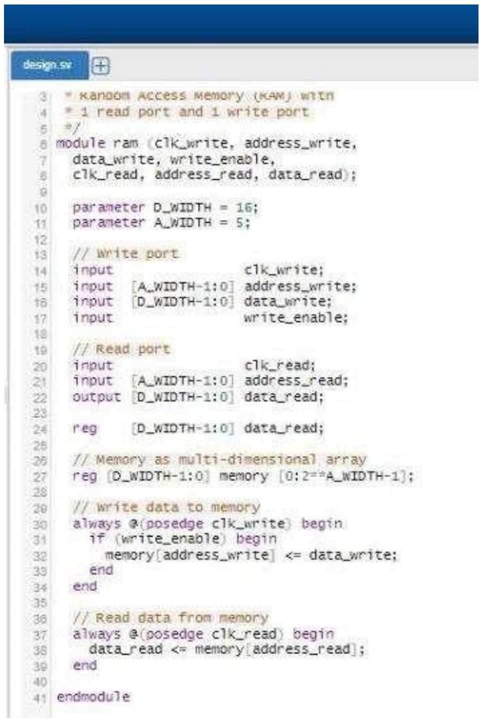
Software Environment:

The software requirements define the system. There should be a definition and a set of requirements. On top of the software requirements, the software requirements specification is constructed. It is useful for estimating costs, planning team activities, carrying out tasks, and keeping track of the team's advancement during the development activity.

- **Language | Verilog & Python 3.x**
- **IDE | Any code editor**
- **OS | Windows 10, Ubuntu 18.0, MacOS 10.12.6**

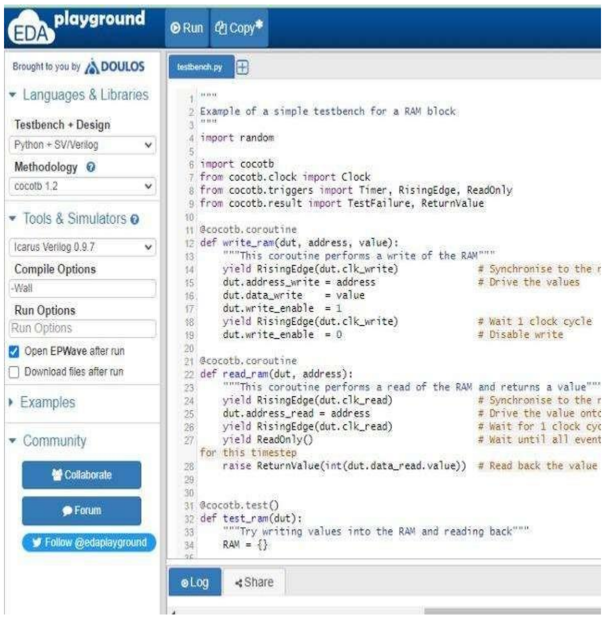
Screenshot:

RTL Screenshot:



```
 * Random Access Memory (RAM) with
 * 1 read port and 1 write port
 */
module ram (clk_write, address_write,
  data_write, write_enable,
  clk_read, address_read, data_read);

  parameter D_WIDTH = 16;
  parameter A_WIDTH = 5;

  // Write port
  input                    clk_write;
  input  [A_WIDTH-1:0]     address_write;
  input  [D_WIDTH-1:0]     data_write;
  input                    write_enable;

  // Read port
  input                    clk_read;
  input  [A_WIDTH-1:0]     address_read;
  output [D_WIDTH-1:0]     data_read;

  reg    [D_WIDTH-1:0]     data_read;

  // Memory as multi-dimensional array
  reg [D_WIDTH-1:0] memory [0:2**A_WIDTH-1];

  // Write data to memory
  always @(posedge clk_write) begin
    if (write_enable) begin
      memory[address_write] <= data_write;
    end
  end

  // Read data from memory
  always @(posedge clk_read) begin
    data_read <= memory[address_read];
  end

endmodule
```

Table Top Module:

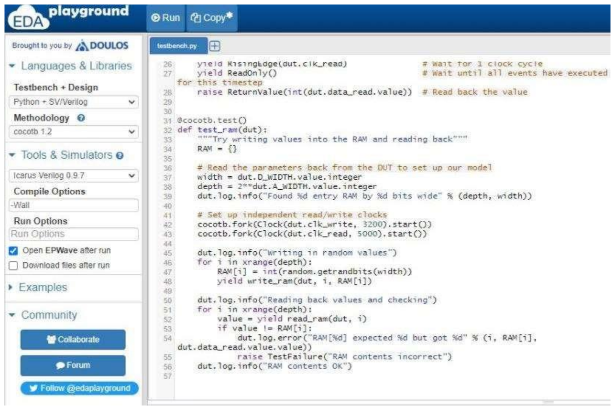


Testbench + Design: Python + SV/Verilog
Methodology: cocotb 1.2
Tools & Simulators: Icarus Verilog 0.9.7
Compile Options: -Wall
Run Options
☑ Open EPWave after run
☐ Download files after run

```
"""
Example of a simple testbench for a RAM block
"""
import random

import cocotb
from cocotb.clock import Clock
from cocotb.triggers import Timer, RisingEdge, ReadOnly
from cocotb.result import TestFailure, ReturnValue

@cocotb.coroutine
def write_ram(dut, address, value):
    """This coroutine performs a write of the RAM"""
    yield RisingEdge(dut.clk_write)              # Synchronise to the r
    dut.address_write = address                  # Drive the values
    dut.data_write    = value
    dut.write_enable  = 1
    yield RisingEdge(dut.clk_write)              # Wait 1 clock cycle
    dut.write_enable  = 0                        # Disable write

@cocotb.coroutine
def read_ram(dut, address):
    """This coroutine performs a read of the RAM and returns a value"""
    yield RisingEdge(dut.clk_read)               # Synchronise to the r
    dut.address_read = address                   # Drive the value onto
    yield RisingEdge(dut.clk_read)               # Wait for 1 clock cyc
    yield ReadOnly()                             # Wait until all event
for this timestep
    raise ReturnValue(int(dut.data_read.value))  # Read back the value


@cocotb.test()
def test_ram(dut):
    """Try writing values into the RAM and reading back"""
    RAM = {}
```

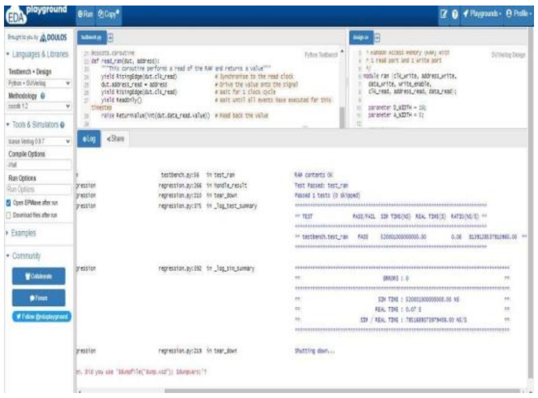
Verification result:

COCOTB module:



Testbench + Design: Python + SV/Verilog
Methodology: cocotb 1.2
Tools & Simulators: Icarus Verilog 0.9.7
Compile Options: -Wall
Run Options
☑ Open EPWave after run
☐ Download files after run

```
    yield RisingEdge(dut.clk_read)               # Wait for 1 clock cycle
    yield ReadOnly()                             # Wait until all events have executed
for this timestep
    raise ReturnValue(int(dut.data_read.value))  # Read back the value


@cocotb.test()
def test_ram(dut):
    """Try writing values into the RAM and reading back"""
    RAM = {}

    # Read the parameters back from the DUT to set up our model
    width = dut.D_WIDTH.value.integer
    depth = 2**dut.A_WIDTH.value.integer
    dut.log.info("Found %d entry RAM by %d bits wide" % (depth, width))

    # Set up independent read/write clocks
    cocotb.fork(Clock(dut.clk_write, 3200).start())
    cocotb.fork(Clock(dut.clk_read, 5000).start())

    dut.log.info("Writing in random values")
    for i in xrange(depth):
        RAM[i] = int(random.getrandbits(width))
        yield write_ram(dut, i, RAM[i])

    dut.log.info("Reading back values and checking")
    for i in xrange(depth):
        value = yield read_ram(dut, i)
        if value != RAM[i]:
            dut.log.error("RAM[%d] expected %d but got %d" % (i, RAM[i],
dut.data_read.value.value))
            raise TestFailure("RAM contents incorrect")
    dut.log.info("RAM contents OK")
```



```
@cocotb.coroutine
def read_ram(dut, address):
    """This coroutine performs a read of the RAM and returns a value"""
    yield RisingEdge(dut.clk_read)               # Synchronise to the read clock
    dut.address_read = address                   # Drive the value onto the signal
    yield RisingEdge(dut.clk_read)               # Wait for 1 clock cycle
    yield ReadOnly()                             # Wait until all events have executed for this
timestep
    raise ReturnValue(int(dut.data_read.value))  # Read back the value
```

```
 * Random Access Memory (RAM) with
 * 1 read port and 1 write port
 */
module ram (clk_write, address_write,
  data_write, write_enable,
  clk_read, address_read, data_read);

  parameter D_WIDTH = 16;
  parameter A_WIDTH = 5;
```

```
            testbench.py:56   in test_ram               RAM contents OK
gression    regression.py:266 in handle_result          Test Passed: test_ram
gression    regression.py:210 in tear_down              Passed 1 tests (0 skipped)
gression    regression.py:375 in _log_test_summary      **************************************************************
                                                        ** TEST              PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
                                                        **************************************************************
                                                        ** testbench.test_ram   PASS    5200010000000000.00    0.08   [illegible] **
                                                        **************************************************************

gression    regression.py:392 in _log_sim_summary       ******************************************************************
                                                        **                       ERRORS : 0                             **
                                                        ******************************************************************
                                                        **         SIM TIME : 5200010000000000.00 NS                    **
                                                        **                REAL TIME : 0.07 S                            **
                                                        **      SIM / REAL TIME : [illegible] NS/S                      **
                                                        ******************************************************************

gression    regression.py:219 in tear_down             Shutting down...

n. Did you use '$dumpfile("dump.vcd"); $dumpvars;'?
```

CONCLUSION

So, by using our application, we can design and verify RAM at a low cost. through the verification process based on Python. There is no additional RTL code required to use Cocotb. The DUT is instantiated at the top level in the simulator. Python is employed to stimulate the DUT's inputs and track its results. It can be quite helpful to folks who are unfamiliar with it because it does not require understanding of HDLs. The simulation time for the IP cores verified using the standard UVM methodology compared to the cocotb framework has significantly improved because the python-based framework only needs the c-model to generate the vectors, as opposed to the UVM methodology, which needs both the RTL and the c-model to verify the design.

REFERENCES

1. Brian Towles & William J. Dally (2001) Route Packets Not Wires on Chip Interconnection Network, Proceedings of the 38th Annual Design Automation Conference, pp. 684-689.
2. Sanju V & Niranjan N Chiplunkar (2008) . Proceedings of the International Conference on Emerging Methods in Computing, Electronics, Embedded System & VLSI Design, "Network-on- Chip: A Short Overview"
3. C. Batten & S. Jiang, P. Pan, Y. Ou IEEE Micro, vol. 40, no. 4, pp. 58–66, 1 July–2 August2020; doi:10.1109/MM .2020.2997638; "PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification."
4. C.Spear, second version of "System Verilog for Verification: A Guide to Learning the Testbench Language Features," published in 2008.
5. Martin Spicknel, Muhammad AlI and Michael Well (2008) The 4th European Conference on Circuits and Systems for Communications' "Networks on Chip: Scalable Interconnects for Future Systems on Chip" was published.
6. Wim Heirman, Karel Bruneel, Robbe Vancayseele, Brahim Al Farisi, and Dirk Stroobandt (2011) 6th International Workshop on Reconfigurable. Communication-Centric Systems-on-Chip, "RecoNoC: A Reconfigurable Network-on-Chip" (ReCoSoC).
7. Abd El Ghany, Salma Hesham, Jens Rettkowski and Diana Göhringer, Mohamed A (2015) Springer International Publishing Switzerland, "Study on Real-Time Network-on-Chip Architectures."
8. Niranjan Chiplunkar, Koushika C, Sanju V, Sharmili R., and M. Khalid (2014) International Journal of Computational Science and Engineering, No. 9, "Design and implementation of a network on chip-based simulator: a performance analysis."
9. Slim Ben Saoud and Ahmed Ben, Abdallah (2013) International Journal of Advanced Computer Science and Applications, No. 4, "A Review of Network-On-Chip Tools."
10. Razvan Nane, Fabrizio Ferrand, Yu Ting Chen and Jongsok Choi (2016) IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, Vol. 35, "A Study and Assessment of FPGA High-Level Synthesis Tools."