Installing Verilator on Ubuntu /Installing Verilator Using Binary Release

Created by: Rao Muhammad Umer

Download and Extract Binary Release

- Download the OSS CAD Suite binary release package:
- Command: `wget https://github.com/YosysHQ/oss-cad-suite-build/releases/download/2024-04-09/oss-cad-suite-linux-x64-20240409.tgz`
- Create a directory named "Utils" in the user's home directory:
- Command: `mkdir ~/Utils`
- Extract the downloaded package into the "Utils" directory:
- Command: `tar xzf oss-cad-suite-linux-x64-20240409.tgz -C ~/Utils`

Verify Verilator Installation

- Execute Verilator to verify its installation:
- Command: /home/ubuntu/Utils/oss-cad-suite/bin/verilator
- Check the version of the installed Verilator:
- Command: /home/ubuntu/Utils/oss-cad-suite/bin/verilator --version

Update PATH Environment Variable

- Open the .bashrc file for editing:
- Command: `nano ~/.bashrc`
- List the Verilator executable file's permissions and location:
- Command: `ls -l ~/Utils/oss-cad-suite/bin/verilator`
- Add Verilator binary directory to the PATH environment variable:
- Command: `export PATH=${PATH}:/home/ubuntu/Utils/oss-cad-suite/bin/`
- Display the updated PATH variable & Verify the default shell:
- Command: `echo $PATH` & `echo $SHELL`

Edit .bashrc Configuration & Final Verification

- Open the .bashrc file in a text editor for modification of the path of verilator :
- Command: `code ~/.bashrc`   // open the vscode on bashrc file
- Add this path to the bashrc file and save the file and exit.
- `export PATH=${PATH}:/home/ubuntu/Utils/oss-cad-suite/bin/`
- OR: `alias verilator='/home/ubuntu/Utils/oss-cad-suite/bin/verilator'`
- Run Verilator to ensure it's accessible after updating the PATH:
- Command: `verilator`
- Check the version of Verilator again : `verilator --version`

Installing and Using GTKWave

Install GTKWave :

- Install GTKWave, a waveform viewer for digital simulation traces:
- Command: sudo apt-get install gtkwave

View Simulation Waveforms :

- Open GTKWave and load the specified waveform file for visualization:
- Command: gtkwave OR gtkwave <waveform_file>

GTKWave Commands

- Check the version of installed GTKWave:
- Command: gtkwave --version
- Display help information about GTKWave and its usage:
- Command: gtkwave --help
- Access the GTKWave manual for detailed documentation:
- Command: man gtkwave

Explanation of Verilator Flags

- --binary:
- Flag: --binary
- Explanation: Specifies that Verilator should generate a binary executable for the simulation. This means that Verilator compiles the specified Verilog/SystemVerilog files into a binary executable file that can be run to perform the simulation.
- --timing:
- Flag enabling timing simulation. When specified, Verilator will perform timing-accurate simulation, considering the delays specified in the Verilog/SystemVerilog design. This is useful for simulations where timing behavior is important, such as when verifying clocked designs.

Explanation of Verilator Flags

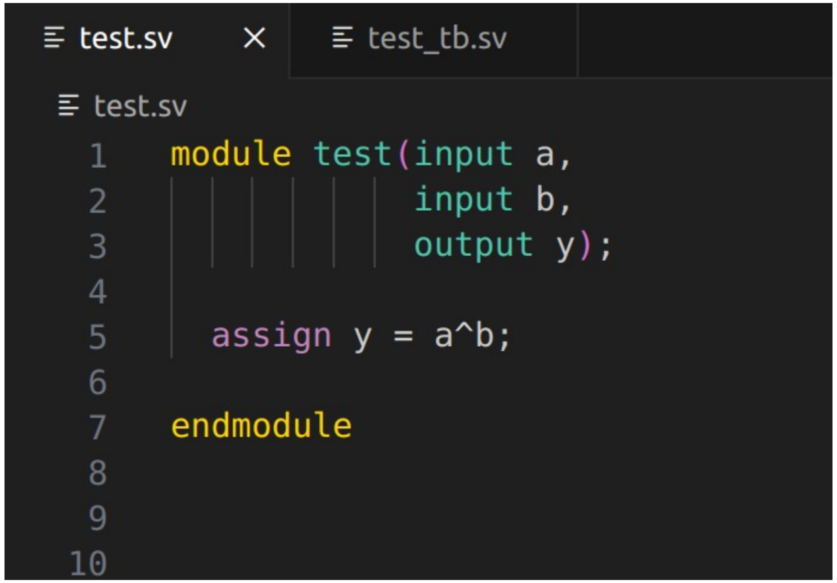
- --cc:
- Flag: --cc
- Explanation: Instructs Verilator to generate C++ code for the simulation. Verilator generates C++ code from the specified Verilog/SystemVerilog files, allowing users to compile the generated code using a C++ compiler.
- --trace:
- Flag: --trace
- Explanation: Enables tracing during the simulation. Tracing records the activity and behavior of signals and variables in the design during simulation. It generates a VCD (Value Change Dump) file, which can be used with waveform viewers like GTKWave to visualize and analyze the simulation results.

Explanation of Verilator Flags

- --exe:
- Flag: --exe
- Explanation: Specifies that Verilator should generate an executable file for the simulation. Verilator compiles the generated C++ code into an executable file that can be run to perform the simulation.
- -o simulation:
- Option: -o simulation
- Explanation: Specifies the name of the output file for the Verilator simulation. In this case, -o simulation indicates that the output executable file should be named "simulation". This option allows users to specify a custom name for the simulation executable generated by Verilator.

Simulating and Viewing Waveforms with Verilator and GTKWave

- Open the the editor create the design file.sv and design_testbench.sv
-



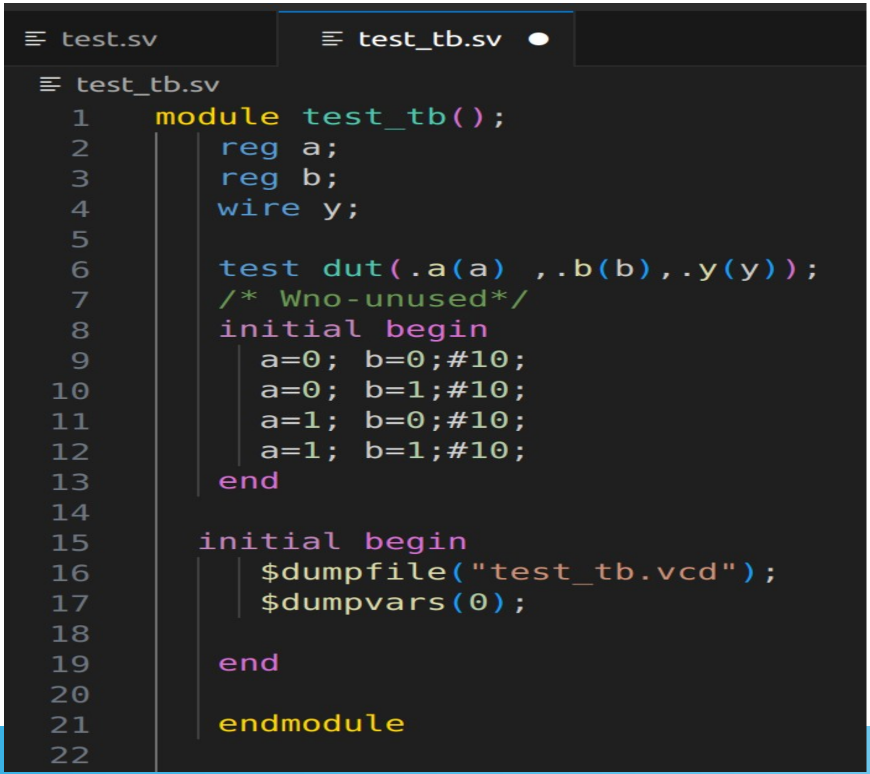
test.sv | test_tb.sv

test.sv

```
module test(input a,
            input b,
            output y);

  assign y = a^b;

endmodule
```



test.sv | test_tb.sv

test_tb.sv

```
module test_tb();
   reg a;
   reg b;
   wire y;

   test dut(.a(a) ,.b(b),.y(y));
   /* Wno-unused*/
   initial begin
     a=0; b=0;#10;
     a=0; b=1;#10;
     a=1; b=0;#10;
     a=1; b=1;#10;
   end

  initial begin
     $dumpfile("test_tb.vcd");
     $dumpvars(0);

   end

   endmodule
```

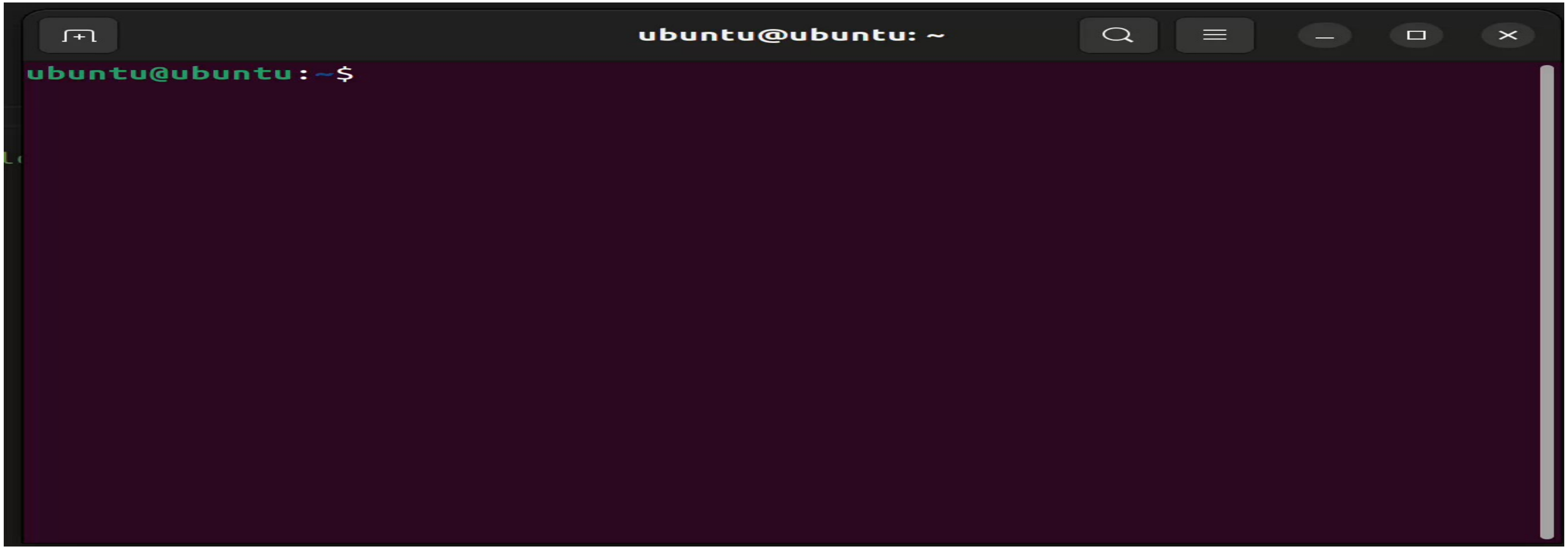
Simulating and Viewing Waveforms with Verilator and GTKWave

Now , use this commad to run your design and testbench in terminal:

- Command: cd Desktop/linux_commad/vscode/sample_test/
- Command: verilator --binary test.sv test_tb.sv --timing –trace
- Command: cd ./obj_dir/
- Command: ls   // in this time dump.vcd file not created
- Command: ./Vtest   // after run this file you see the dump.vcd file
- Command: gtkwave 'test_tb.vcd'

now, see the video for this commad in next slide.

Simulating and Viewing Waveforms with Verilator and GTKWave



```
ubuntu@ubuntu:~$
```

Simulating and Viewing Waveforms with Verilator and GTKWave

- This commad is optional if you use this methods also work fines.
- 1st Step:
- Command: cd Desktop/linux_commad/vscode/sample_test/
- Explanation: Change directory to the specified path using the provided terminal shortcut. This command navigates to the directory where Verilog/SystemVerilog files and simulations are located.
- 2nd Step:
- Command: verilator --binary test.sv test_tb.sv --top test_tb
- Explanation: Compile the Verilog/SystemVerilog files test.sv and test_tb.sv using Verilator. The --top flag specifies the top-level module as test_tb.

Simulating and Viewing Waveforms with Verilator and GTKWave

- 3rd Step:
- Command: verilator --binary --cc test.sv test_tb.sv --trace --exe -o simulation
- Explanation: Compile the Verilog/SystemVerilog files test.sv and test_tb.sv with tracing enabled, creating an executable named simulation. This command prepares the Verilog/SystemVerilog files for simulation.
- 4th Step:
- Command: cd ./obj_dir/
- Explanation: Change directory to the obj_dir where Verilator output files are stored. This directory typically contains the compiled object files generated during the Verilator compilation process.

Simulating and Viewing Waveforms with Verilator and GTKWave

- 5th Step:
- Command: ls
- Explanation: List the files in the obj_dir directory. This command is used to verify the contents of the directory, ensuring that the Verilator compilation process generated the necessary output files.
- 6th Step:
- Command: ./out.vvp
- Explanation: Run the Verilog simulation executable out.vvp. This command executes the compiled Verilog/SystemVerilog simulation, initiating the simulation process.

Simulating and Viewing Waveforms with Verilator and GTKWave

- 7th Step:
- Command: ./simulation
- Explanation: Execute the simulation program. This command runs the compiled simulation program generated by Verilator, allowing for the simulation of the Verilog/SystemVerilog design.
- Final Step:
- Command: gtkwave   OR   gtkwave 'dump.vcd'
- Explanation: Open GTKWave to visualize simulation waveforms. GTKWave is a waveform viewer that allows users to analyze and visualize simulation results, making it easier to debug and understand the behavior of digital designs.

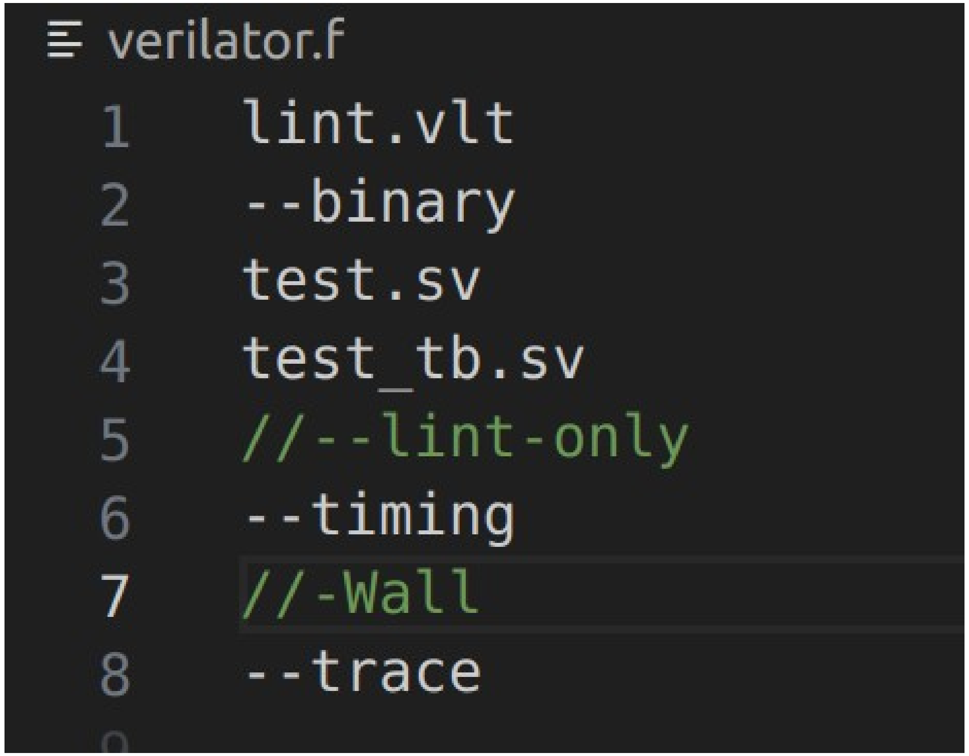
Simulating and Viewing Waveforms with Verilator and GTKWave

You can also create the commad file :

- Open editor create the .f extension file.

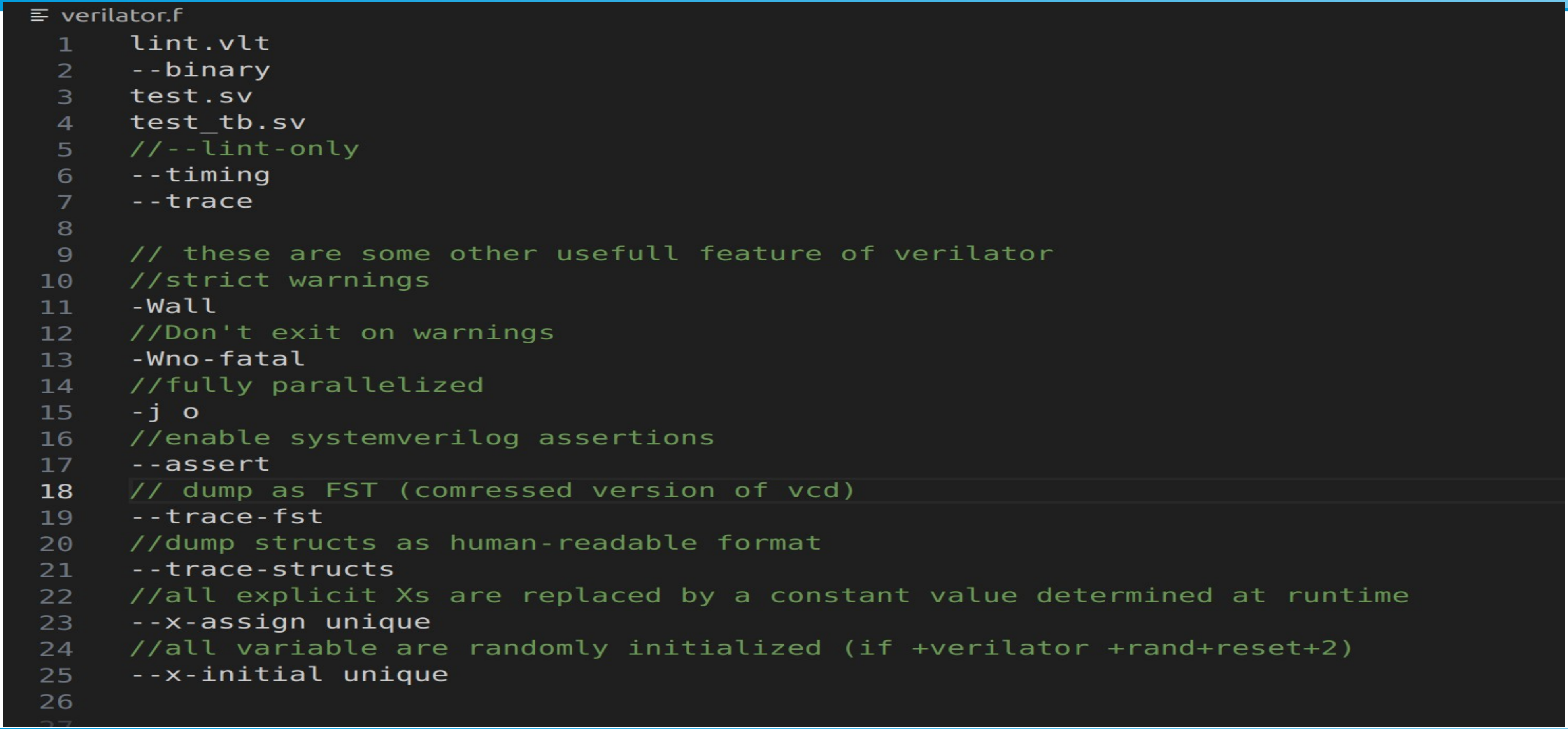
Run commad for this file :

- verilator -f verilator.f



verilator.f

```
lint.vlt
--binary
test.sv
test_tb.sv
//--lint-only
--timing
//-Wall
--trace
```



verilator.f

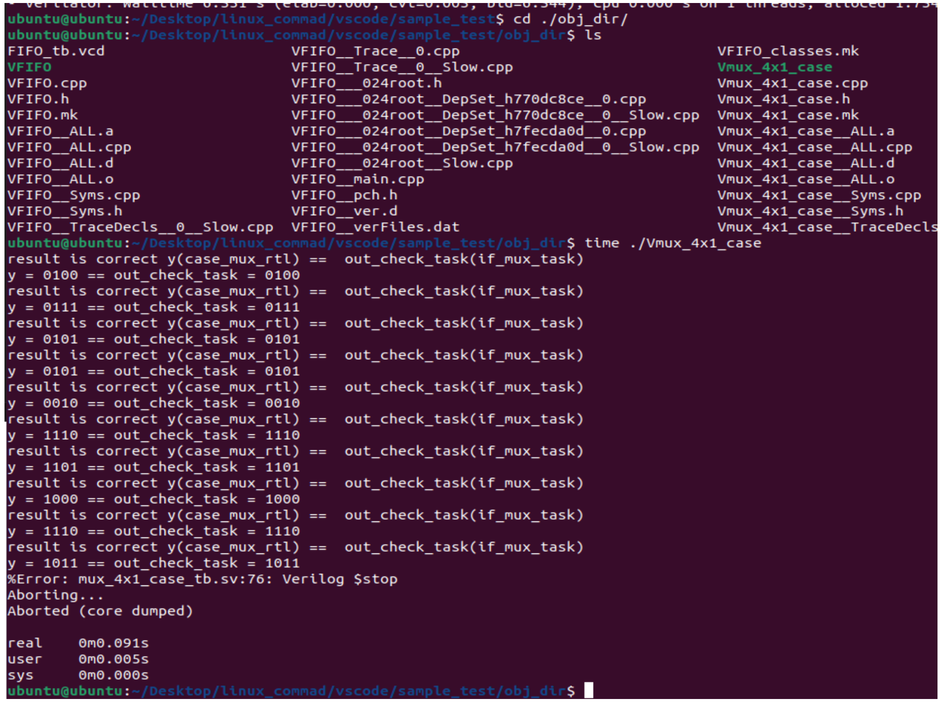
```
lint.vlt
--binary
test.sv
test_tb.sv
//--lint-only
--timing
--trace

// these are some other usefull feature of verilator
//strict warnings
-Wall
//Don't exit on warnings
-Wno-fatal
//fully parallelized
-j 0
//enable systemverilog assertions
--assert
// dump as FST (comressed version of vcd)
--trace-fst
//dump structs as human-readable format
--trace-structs
//all explicit Xs are replaced by a constant value determined at runtime
--x-assign unique
//all variable are randomly initialized (if +verilator +rand+reset+2)
--x-initial unique

```

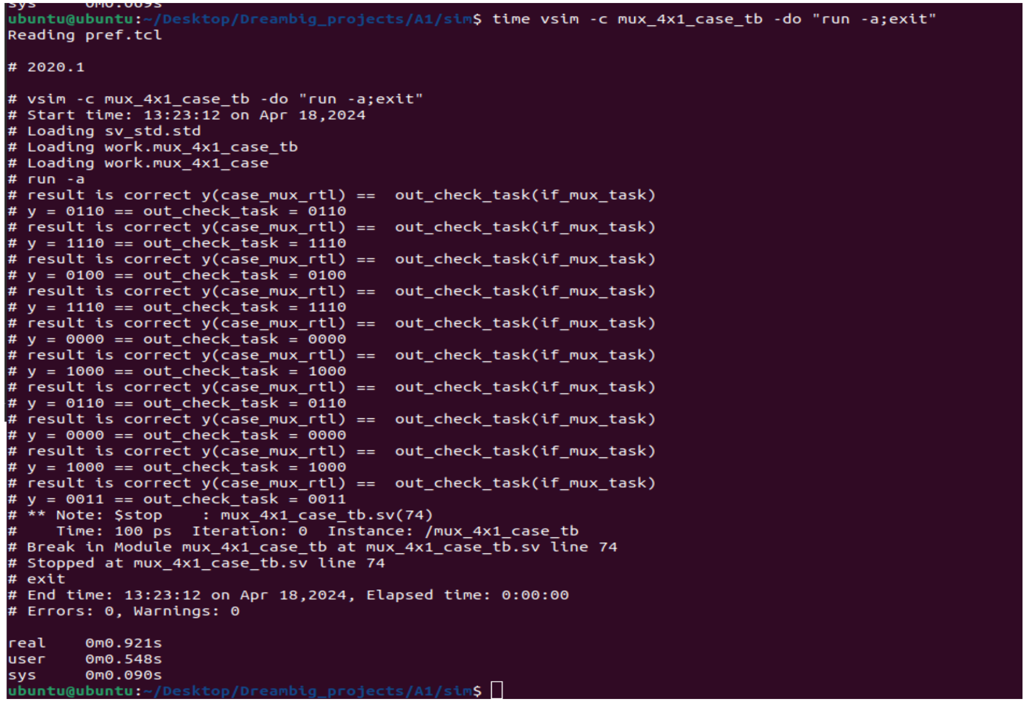
Simulation time comparison

Verilator Vs Modelsim



```
ubuntu@ubuntu:~/Desktop/linux_commad/vscode/sample_test$ cd ./obj_dir/
ubuntu@ubuntu:~/Desktop/linux_commad/vscode/sample_test/obj_dir$ ls
FIFO_tb.vcd                     VFIFO__Trace__0.cpp                          VFIFO_classes.mk
VFIFO                           VFIFO__Trace__0__Slow.cpp                    Vmux_4x1_case
VFIFO.cpp                       VFIFO___024root.h                            Vmux_4x1_case.cpp
VFIFO.h                         VFIFO___024root__DepSet_h770dc8ce__0.cpp     Vmux_4x1_case.h
VFIFO.mk                        VFIFO___024root__DepSet_h770dc8ce__0__Slow.cpp  Vmux_4x1_case.mk
VFIFO__ALL.a                    VFIFO___024root__DepSet_h7fecda0d__0.cpp     Vmux_4x1_case__ALL.a
VFIFO__ALL.cpp                  VFIFO___024root__DepSet_h7fecda0d__0__Slow.cpp  Vmux_4x1_case__ALL.cpp
VFIFO__ALL.d                    VFIFO___024root__Slow.cpp                    Vmux_4x1_case__ALL.d
VFIFO__ALL.o                    VFIFO__main.cpp                              Vmux_4x1_case__ALL.o
VFIFO__Syms.cpp                 VFIFO__pch.h                                 Vmux_4x1_case__Syms.cpp
VFIFO__Syms.h                   VFIFO__ver.d                                 Vmux_4x1_case__Syms.h
VFIFO__TraceDecls__0__Slow.cpp  VFIFO__verFiles.dat                          Vmux_4x1_case__TraceDecls
ubuntu@ubuntu:~/Desktop/linux_commad/vscode/sample_test/obj_dir$ time ./Vmux_4x1_case
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 0100 == out_check_task = 0100
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 0111 == out_check_task = 0111
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 0101 == out_check_task = 0101
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 0101 == out_check_task = 0101
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 0010 == out_check_task = 0010
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 1110 == out_check_task = 1110
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 1101 == out_check_task = 1101
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 1000 == out_check_task = 1000
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 1110 == out_check_task = 1110
result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
y = 1011 == out_check_task = 1011
%Error: mux_4x1_case_tb.sv:76: Verilog $stop
Aborting...
Aborted (core dumped)

real    0m0.091s
user    0m0.005s
sys     0m0.000s
ubuntu@ubuntu:~/Desktop/linux_commad/vscode/sample_test/obj_dir$
```



```
ubuntu@ubuntu:~/Desktop/Dreambig_projects/A1/sim$ time vsim -c mux_4x1_case_tb -do "run -a;exit"
Reading pref.tcl

# 2020.1

# vsim -c mux_4x1_case_tb -do "run -a;exit"
# Start time: 13:23:12 on Apr 18,2024
# Loading sv_std.std
# Loading work.mux_4x1_case_tb
# Loading work.mux_4x1_case
# run -a
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 0110 == out_check_task = 0110
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 1110 == out_check_task = 1110
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 0100 == out_check_task = 0100
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 1110 == out_check_task = 1110
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 0000 == out_check_task = 0000
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 1000 == out_check_task = 1000
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 0110 == out_check_task = 0110
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 0000 == out_check_task = 0000
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 1000 == out_check_task = 1000
# result is correct y(case_mux_rtl) ==  out_check_task(if_mux_task)
# y = 0011 == out_check_task = 0011
# ** Note: $stop    : mux_4x1_case_tb.sv(74)
#    Time: 100 ps  Iteration: 0  Instance: /mux_4x1_case_tb
# Break in Module mux_4x1_case_tb at mux_4x1_case_tb.sv line 74
# Stopped at mux_4x1_case_tb.sv line 74
# exit
# End time: 13:23:12 on Apr 18,2024, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0

real    0m0.921s
user    0m0.548s
sys     0m0.090s
ubuntu@ubuntu:~/Desktop/Dreambig_projects/A1/sim$
```

- **Verilator run simulations faster as compare to Modelsim**

- ModelSim:
- Supports four-state variables (0, 1, X, Z).
- Suitable for designs involving VHDL or Verilog with extensive use of four-state logic.
- Provides comprehensive support for both VHDL and Verilog languages.
- Offers a range of simulation modes and advanced debugging tools.
- Commercial product with licensing requirements.
- Verilator:
- Primarily deals with two-state logic (0, 1).
- Optimized for synthesizable RTL code simulation.
- May not directly support four-state variables like X and Z.
- Open-source tool under the GNU LGPL, freely available.
- Known for high simulation speed, particularly for large designs.

THANKYOU ….