Module-1 Training - Lab Manual: Markdown, Makefiles, and Verilator

This lab manual delves into Markdown Language, Makefiles for automating compilation, and verilator for simulating system verilog design files. Remember, consistent practice and exploration of additional resources are key to mastering these skills.

Markdown

(Percentage: 10%)

MEDS Blog: [click here]

Exercises:

1. Add a readme.md file for your github repository
2. Add center aligned figures
3. Add a checklist of your completed topics in the training
4. Write a shell scripting tutorial

Makefiles

(Percentage: 30%)

Makefiles are configuration files that automate the compilation process for software projects. They specify dependencies between source files and how to generate the final executable.

Learning Resources:

- Hand Book: (available here)

- MEDS slides: [available here]
- The GNU Make Manual: (available here)
- Makefile Tutorial: (available here)
- MIT Make File Guide: (available here)

Exercises:

1. **Basic Makefile Structure:**
    - Understand the syntax of Makefiles, including target definitions, prerequisites, and recipes (commands to be executed).
2. **Compiling C Programs with Makefiles:**
    - Create a Makefile that specifies how to compile your C code using gcc. Define targets for compiling individual source files and building the final executable.
3. **Dependencies and Automatic Builds:**
    - Learn how to leverage dependencies in Makefiles to ensure only modified files are recompiled during subsequent builds.
4. **Advanced Makefile Features:**
    - Explore features like wildcards, variables, and conditional statements for more complex build scenarios.

Tip: Start with simple Makefiles for single-file C programs and gradually add complexity as you understand the concepts. Online tutorials with examples can be helpful for practicing Makefile creation.

Verilator

(Percentage: 40%)

Verilator is a software tool that translates Verilog code into executable C++ code. This C++ code can then be compiled and simulated using a standard C++ compiler.

Learning Resources:

- Verilator Documentation: [available here]
- Getting Started with Verilator: [available here]
- MEDS Blog: [available here]
- Installation: [available here]
- Usage: [click here]

Exercises:

1. **Installing Verilator:**
    - Follow the official Verilator documentation for installation instructions on your chosen platform.
2. **Simulating a Simple Verilog Module:**
    - Write a basic Verilog module (e.g., an adder) and use Verilator to translate it into C++ code.
    - Compile the generated C++ code and run the simulation to verify the module's functionality.
3. **Advanced Verilator Features:**
    - Explore Verilator options for specifying testbench files, generating waveform traces, and customizing simulation behavior.

Benefits of Verilator:

- **Platform Independence:** Verilator allows simulation on various platforms by translating Verilog to C++.

- **Faster Simulation Speed:** C++ simulation can sometimes be faster than native Verilog simulators.
- **Integration with C++ Code:** Verilator enables easier integration of Verilog designs with C++ testbenches or environments.

GTKWave

(Percentage: 20%)

GTKWave is a graphical waveform viewer commonly used to visualize and analyze the simulation results generated by Verilator or Icarus Verilog.

Learning Resources:

- GTKWave Documentation: [here]
- GTKWave Tutorials: [here], [here]

Exercises:

7. **Installing GTKWave:**
    - Follow the official GTKWave documentation for installation instructions on your chosen platform.
8. **Loading Waveform Traces:**
    - Generate waveform traces (VCD format) from your Verilator or Icarus Verilog simulations.
    - Open the VCD file in GTKWave to visualize the signals and their behavior over time.
9. **Zooming and Analysis:**

- Explore GTKWave's features for zooming in/out on specific time regions, adding cursors for measurements, and analyzing signal transitions.

Benefits of GTKWave:

- **Intuitive Visualization:** Provides a user-friendly interface to visualize complex Verilog simulation results.
- **Signal Analysis:** Enables detailed analysis of signal behavior and timing relationships.
- **Debugging Aid:** Assists in debugging Verilog designs by visually identifying errors or unexpected behavior.

Combining these tools effectively streamlines your Verilog development workflow:

1. **Write Verilog code:** Design your digital circuits using Verilog.
2. **Simulate with Verilator:** Simulate the design to verify functionality and identify potential issues.
3. **Visualize results with GTKWave:** Analyze the simulation waveforms to gain deeper insights into signal behavior.

By mastering these tools, you'll be well-equipped to tackle digital design challenges using Verilog effectively.