UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ÍCARO DANTAS DE ARAÚJO LIMA

SCALABLE WEB-BASED FPGA BOARD SIMULATOR

CAMPINA GRANDE - PB

2021

ÍCARO DANTAS DE ARAÚJO LIMA

SCALABLE WEB-BASED FPGA BOARD SIMULATOR

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Elmar Uwe Kurt Melcher.

CAMPINA GRANDE - PB

2021

L732s Lima, Ícaro Dantas de Araújo.
Scalable web-based FPGA board simulator. / Ícaro Dantas de Araújo Lima. – 2021.

9 f.

Orientador: Prof. Dr. Elmar Uwe Kurt Melcher.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Simulador de placa FPGA. 2. Código em systemverilog. 3. Field-programable gate array - FPGA. 4. Aprendizagem de HDLs. 5. Linguagens de descrição de hardware - HDL. I. Melcher, Elmar Uwe Kurt. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

ÍCARO DANTAS DE ARAÚJO LIMA

SCALABLE WEB-BASED FPGA BOARD SIMULATOR

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Elmar Uwe Kurt Melcher

Orientador – UASC/CEEI/UFCG

Professor Dr. Thiago Emmanuel Pereira da Cunha Silva

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 20 de Outubro de 2021.

CAMPINA GRANDE - PB

RESUMO (ABSTRACT)

Os métodos de aprendizagem de HDLs (linguagens de descrição de hardware) incluem principalmente a prática com placas reprogramáveis e simuladores. Os maiores obstáculos para o aprendizado são o custo dessas placas, a interface hostil desses simuladores e, às vezes, a tediosa configuração do ambiente, necessária até mesmo para executar uma única linha de código. Este trabalho apresenta um simulador de placa FPGA (field-programmable gate array) baseado em web. O sistema é composto por 2 componentes principais: um front-end e um back-end, seguindo uma arquitetura de microsserviços. É possível escrever código em SystemVerilog e interagir com ele usando uma placa FPGA virtual, exigindo apenas um navegador e acesso à internet. As etapas envolvidas entre a submissão do código do usuário e a simulação, são duas conversões de código. Uma vez que essas conversões podem ser executadas em uma única tarefa, o sistema pode ser escalado horizontalmente. Graças aos eventos enviados pelo servidor e um emulador de console, o usuário pode ver tudo o que está acontecendo nessas tarefas em tempo real.

Scalable Web-Based FPGA Board Simulator

Ícaro Lima
Federal University of Campina Grande
Campina Grande, Brazil
icaro.lima@ccc.ufcg.edu.br

Elmar Melcher
Federal University of Campina Grande
Campina Grande, Brazil
elmar@computacao.ufcg.edu.br

Joseana Fechine
Federal University of Campina Grande
Campina Grande, Brazil
joseana@computacao.ufcg.edu.br

ABSTRACT

Methods of learning HDLs (hardware description languages) mainly include practice with reprogrammable boards and simulators. The biggest obstacles to learning are the cost of these cards, the unfriendly interface of these simulators, and sometimes the tedious environment setup needed even to run a single line of code. This work presents a web-based FPGA (field-programmable gate array) board simulator. The system is composed of 2 main components: a front-end and a back-end, following a microservices architecture. It is possible to write code in SystemVerilog and interact with it using a virtual FPGA board, requiring only a browser and internet access. The steps involved between the user code input and the simulation are two code conversions. Since these conversions can run on a one-shot task, the system can be scaled horizontally. Thanks to server-sent events and a console emulator, the user can see everything happening on these tasks in real-time.

KEYWORDS

Simulator, FPGA board, HDL, SystemVerilog.

REPOSITORIES

https://github.com/orgs/learn-systemverilog/repositories

1 INTRODUCTION

Knowing logic circuits is critical for electrical and computer engineers [2]. In order to deeply understand the concepts, it is essential to study by reading materials and practicing actively [4].

With the increase in the complexity of logic circuits, HDLs gained more and more relevance. Because due to the growing number of logic gates (e.g., 100,000), it became impossible to design or verify these circuits using manual methods, like paper or breadboards [3].

Reprogrammable FPGAs have been widely used to teach HDLs [2] by allowing students to practice logic circuit design incrementally, similar to software development.

The cost of FPGAs is an impediment for many students, making many educational institutions need to provide one for each student, which in addition to raising the expense, does not solve the problem entirely. Equipment provided by these institutions are usually kept in laboratories for exclusive use during classes, or permissions and additional bureaucracy are imposed to be accessed or taken home. Even those students who can afford to buy an FPGA end up facing problems, including the difficulty in preparing the computing environment, choosing a model, and the need for extra hardware to make these FPGAs stay with a friendlier interface. Having all this ready, it is still necessary for the student to keep cables and/or adapters connecting the FPGA to their computer, which can be inconvenient.

Simulators have been used to solve most of the problems mentioned above. However, most simulators are intended to be used for verification, requiring a testbench to be created in order to be able to simulate a design. Another problem with these simulators is that most of them need to be installed on the student's computer, available only for specific operating systems.

2 SOLUTION

The proposed solution is an FPGA board simulator where the user can write code in SystemVerilog and then interact with it using any device, requiring only a modern web browser with internet access: https://learn-systemverilog.github.io.

2.1 Architecture

The system is composed of two main parts: a front-end and a back-end. The front-end runs on the users' browsers, and the back-end is one or more services whose primary purpose is to convert SystemVerilog code into JavaScript.

Figure 1 shows the currently deployed architecture. Thanks to the stateless nature of the services, a load balancer can be used to distribute user requests.

2.1.1 Server-side vs. Client-side. There were two options to simulate and interact with code written in SystemVerilog. The differences between these two options are similar to the differences between client-side vs. server-side rendering.

The first option would be to leave the simulation running on the server, where it would be necessary to receive commands and continuously report the simulation status back to the front-end. This option has some problems:

- **Resource consumption**: the server needs to maintain at least a process and an open connection for each simultaneous running simulation.
- **Security**: user-submitted code can be malicious.
- **Delay**: network delay between client and server can affect user experience, especially on poor network connections or high-frequency simulations.

The second and chosen option is to simulate the SystemVerilog code in the user's machine, which does not have the problems mentioned above. However, it would be necessary that the code in SystemVerilog somehow be interpreted by the front-end. Fortunately, it is possible to convert code written in SystemVerilog to JavaScript server-side.

Figures 2 and 3 show the sequence of events that occur between client and server during code conversion. Server-Sent Events are used for logging the stdout/stderr/internal messages of the conversion process to the user in real-time. They are also used for

delivering the output JavaScript code if the conversion is successful.

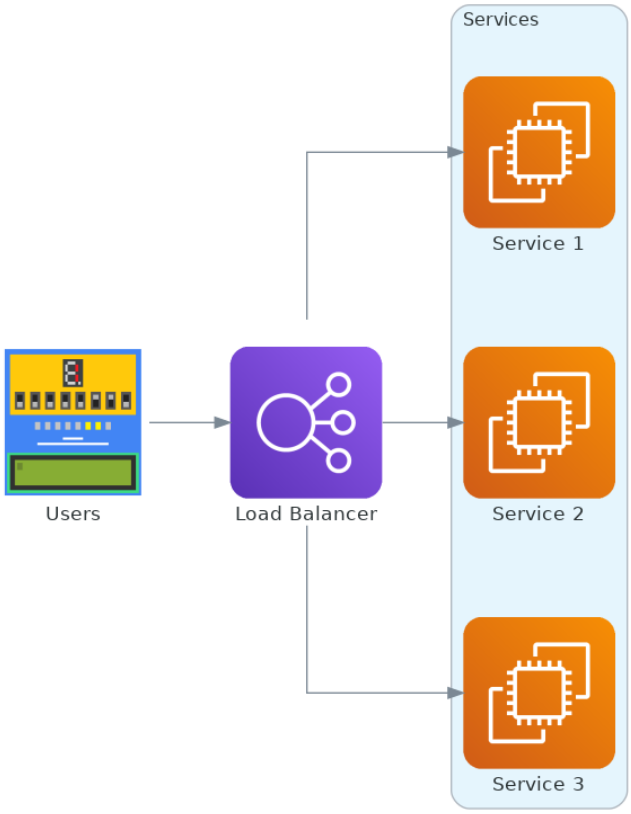

Figure 1: The system architecture

2.1.2 Back-end Workspaces. In order to convert from SystemVerilog to JavaScript, a **workspace** needs to be created for each simultaneous request. These workspaces are temporary folders that isolate output and residuals generated during the process. Each workspace is a copy of the folder `workspace_template`, which contains only two files: `Makefile` and `simulator.cpp`.

The `Makefile` mentioned above helps to simplify commands needed to execute during the conversion.

The `simulator.cpp` is a C++ wrapper file that, when converted to JavaScript, serves as an interface to the front-end to call the simulation itself.

2.2 Technologies

2.2.1 Back-end. The back-end services were implemented using **GoLang** along with **Gin**. GoLang is a fast, lightweight, and widely-used programming language for implementing web servers [5]. Gin is a martini-like API web framework, has good support for Server-Sent Events, and has excellent performance [8].

Verilator and **Emscripten** are used to convert between SystemVerilog and JavaScript. Verilator is a Verilog/SystemVerilog simulator capable of turning SystemVerilog code into a callable C++ code [10]. Emscripten is an open-source compiler toolchain

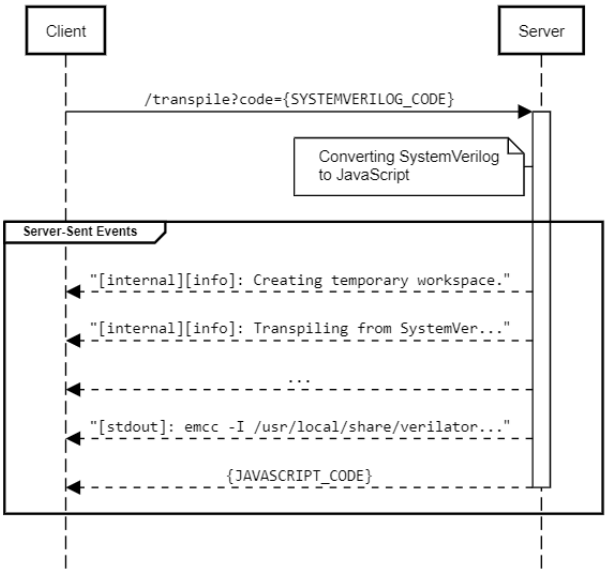

Figure 2: Conversion successful

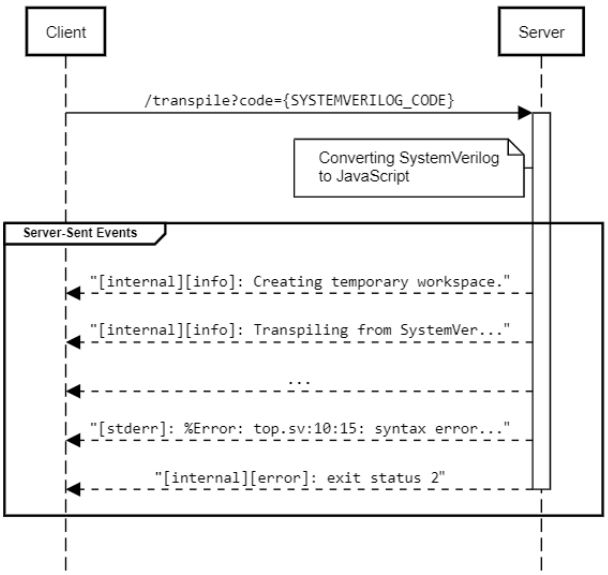

Figure 3: Conversion syntax error

to WebAssembly capable of converting C++ code into JavaScript [11]. Using Verilator followed by Emscripten was one of the few ways found to make this conversion. Previous successful projects supported the decision to use Verilator [6, 7].

2.2.2 Front-end. The web page was implemented using **JavaScript** along with **React** and **PatternFly**. JavaScript is the most well-known scripting language for Web pages. React is a framework for building user interfaces with JavaScript. Patternfly is an open-source design system that can power React, providing standards, tools, and ready-to-use components.

As the web page is simple and has only a few components, JavaScript was enough instead of using a more robust language like **TypeScript**.

The **log-in system** is currently only implemented on the front-end. It uses **React Google Login**, powered by **Google OAuth** [1].

2.3 Code Conversion

Every time the front-end submits a SystemVerilog code through the `/transpile` endpoint of the back-end services, few steps are required to convert this code to JavaScript and send it back to the front-end. Some of these steps are the creation of a workspace and the conversion of SystemVerilog to an intermediate language before turning it into JavaScript. These workspaces are intended to isolate conversions happening at the same time for different requests.

The module responsible for this conversion can be found at `back-end/transpiler/transpiler.go`. It receives the SystemVerilog code as a string and a channel where the results will be reported in real-time.

Below is a more detailed step-by-step on how this conversion happens:

(1) A temporary folder (workspace) with a random name is created, e.g., `/tmp/lsv_api_transpiler_workspace_12345`.
(2) The contents of the folder `workspace_template` are copied to the folder created in the last step.
(3) The SystemVerilog code is converted to C++ by executing the command `make obj_dir`.
(4) The C++ code is converted to JavaScript by executing the command `make simulator.js`.
(5) The workspace is deleted.

For each step, logs are sent to the front-end through the channel and then through Server-Sent Events.

It is valid to mention that the workspace is also deleted if any errors occur during this process.

2.4 Interface

The interface can be broken into 4 main components:

2.4.1 Header. This is a standard page header. It includes the page title, a button that leads to the project repositories, and a sign-in button (Figure 4).

Figure 4: The page header

2.4.2 Simulator. This is an interactive virtual FPGA board. It has 8 switches, 8 LEDs, and a 7-segment display (Figure 5). The user can interact by clicking on the switches, turning them on or off. The LEDs and the 7-segments display are read-only, depending on the simulation to be turned on or off.

2.4.3 Code Editor. This component is a code editor where the user can write code in SystemVerilog. The component also includes three buttons: one to submit the code for simulation (**Simulate!**),

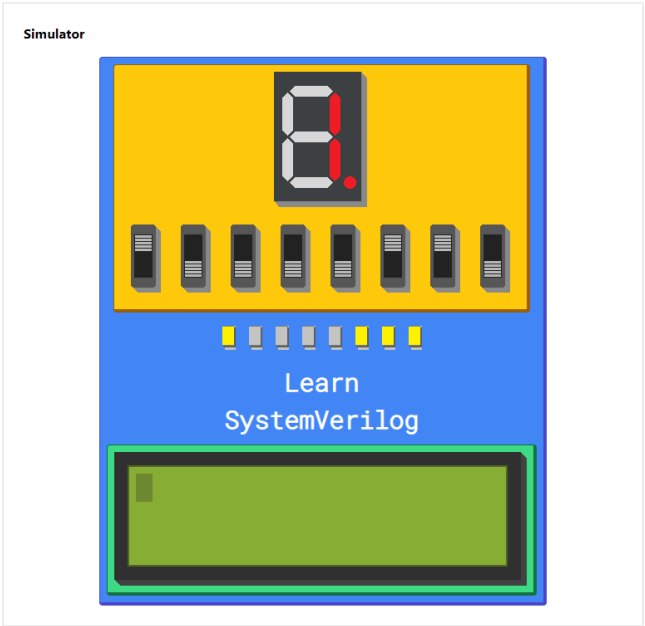

Figure 5: The virtual FPGA board

one to reset the code to the template (**Reset**), and another to reset the code to the last submitted code (**Reload**) (Figure 6).

This code editor is powered by the Monaco Editor (the same used by VS Code), with extra features compared to a simple text editor. Ex: Syntax highlighting [9].

Figure 6: The code editor

2.4.4 Console. This component is a read-only console emulator where the user can watch the stdout and stderr of the procedure needed to start the simulation. Its most common use is to see if any

compilation, connection, or conversion errors happened. It includes a **Clear** button used to erase the console (Figure 7).

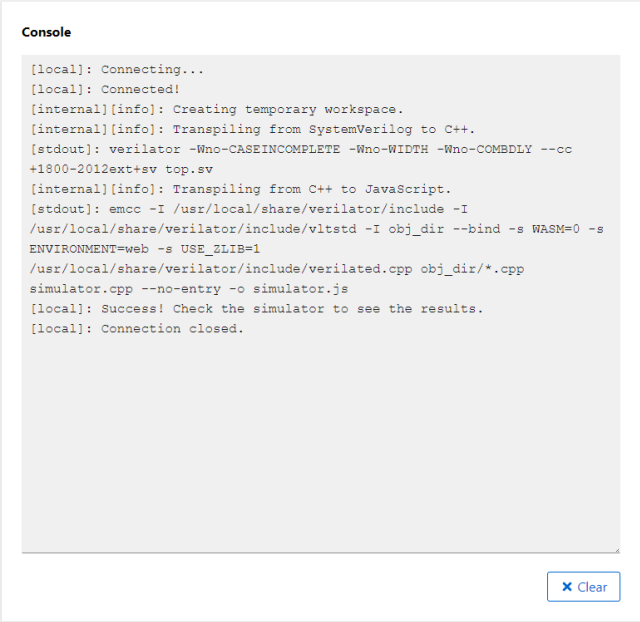

Figure 7: The read-only console emulator

2.5 How to use

Step by step for using the simulator:

(1) Once on the main page, sign in by clicking on the **Sign in** button at the top right corner of the page.
(2) Modify the code in the **Code Editor** section (optional).
(3) Click on the **Simulate!** button.
(4) Watch the **Console**. Wait for the message "*[local]: Success! Check the simulator to see the results.*".
(5) Interact with the simulator in the **Simulator** section.

3 USER EVALUATION

The simulator was tested by 15 users. They were asked to fill out a simple form after completing one or more simulations. The form was presented in Portuguese and the results were translated to English for the purposes of this work. The instructions given to the users were simple: first access the simulator website, then run at least one simulation. All users had already had experience with digital circuits by attending Computer Organization and Architecture classes. The questionnaire had nine questions, eight linear scale questions measured from 1 (bad/totally disagree) to 5 (excellent/totally agree), and one open question for suggestions. Following are the nine questions, in the same order presented to the users:

(1) "How do you rate the simulator's usability as a whole?"
(2) "How do you rate the simulator design as a whole?"
(3) "How do you rate the website's response time?"
(4) "How do you rate the simulator interface response time?"
(5) "How do you assess the waiting time until the start of the simulation?"
(6) "Do you agree that the simulator is intuitive?"
(7) "Do you agree that the simulator is helpful for learning SystemVerilog?"
(8) "Do you agree that the simulator is helpful for learning Computer Organization and Architecture?"
(9) "Space for suggestions:"

Figure 8 shows a column bar chart with the X-axis representing the question number, ranging from 1 to 8, and the Y-axis representing the average user evaluation for that question, ranging from 1 to 5.

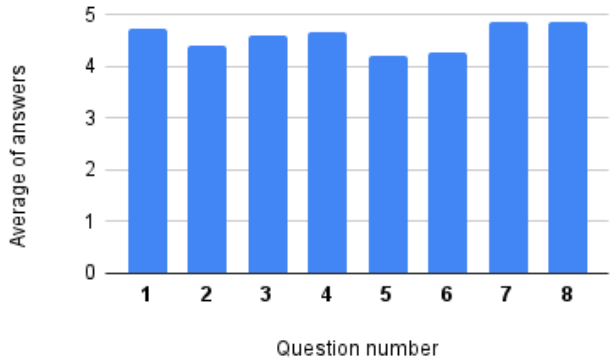

Figure 8: User evaluation

Overall ratings were good, with an average of 4.575. Following are some suggestions collected with question 9:

(1) "Possibility to activate night mode. Allowing visual comfort for those who will use the simulator for many hours."
(2) "Enable clock frequency change via simulator interface."
(3) "Possibility of adding more files to organize the code in modules."
(4) "Remove the need to log in by replacing it with the "I'm not a robot" button."
(5) "Improve the graphical user interface of the web page as a whole, except for the virtual FPGA board itself."

Most suggestions (1, 2, and 5) are punctual and can be resolved on the front-end. Suggestion 4 cannot be attended because it is important to identify when someone tries to send a malicious code. Suggestion 3 is more complex but possible to attend. It needs modifications on the back-end and front-end. An advanced mode can be added to maintain simplicity for those who do not need multiple files.

No users reported technical problems while using the simulator.

4 EXPERIENCE AND LESSONS LEARNED

4.1 Development Process

The first step was to research how it is possible to simulate the code written in SystemVerilog on the front-end, that is, using JavaScript. A direct method for simulating SystemVerilog with JavaScript was not found, but it has been shown that it is possible to simulate the SystemVerilog code in C++ by using Verilator [6, 7, 10].

The next step was to find a way to call that output C++ code in JavaScript. Fortunately, Emscripten makes it possible by converting C++ code into JavaScript.

A test was done successfully to see if it is possible to convert input code written in SystemVerilog and get JavaScript output using the method mentioned above. As Verilator and Emscripten do not run in the browser, a back-end would be needed to do this work.

The architecture was designed so that it could scale to many users. As conversions can be seen as one-shot tasks, it was easy to compare with existing architectures.

It would be necessary to report every event happening in the tasks in real-time to the front-end. There were two leading technologies capable of doing this: WebSockets and Server-Sent Events. Server-Sent Events were used because they are simpler and the events only come in one direction.

At this stage, a prototype of the back-end began to be implemented and tested with Insomnia.

The code started to grow, and a Git repository was created to serve as a version control system. Then, mainly for security reasons (maintain data integrity), the repository was pushed to GitHub.

The last step was to develop a user-friendly web page that could communicate with the back-end and include an animated simulator. A Git repository was also created to the front-end.

4.2 Main Challenges and their Solutions

4.2.1 Client-side SystemVerilog Simulation. Running the simulation client-side is essential to avoid server overload and allow higher clock frequencies. It is necessary that the code in SystemVerilog can somehow be called by the front-end code, written in JavaScript, which is the language used in web pages. At the current time, no library can do this.

The solution was first to "verilate" (convert) the SystemVerilog code to C++ using Verilator, then convert the output code in C++ to JavaScript using Emscripten.

4.2.2 Conversion Feedback. Sometimes converting the SystemVerilog code before simulating can take a long time. Furthermore, the result can be, for example, a syntax error. In these cases, it is essential for the user to receive feedback in real-time.

The solution was logging and sending Server-Sent Events about everything happening server-side and showing them in the front-end's Console section at the same time they arrive on the client-side.

4.3 Limitations and Future Work

4.3.1 Support for other HDLs. The purpose of the simulator is to simulate a virtual FPGA board, not only to simulate SystemVerilog, but currently, it is the only language supported.

Other HDLs can be supported if they can be simulated and called someway by JavaScript. If that is not the case, a simulation running server-side can be considered.

4.3.2 Sandboxing. Workspaces are essential to separate output and residuals during code conversions. Although these workspaces work well, if there is some security issue in any conversion step, users can send malicious code to gain privileged access to the server, steal some data, or use large amounts of resources, even without executing the code server-side.

Instead of using only temporary folders, some sandboxing technology can block any tentative of access outside of the workspace. Furthermore, a sandbox can be used to limit resource usage such as CPU and memory.

4.3.3 Login. Although the user needs to be logged in to perform a simulation, the current login process is not verified by the server. Therefore, malicious users can make requests without being logged in, allowing Denial-of-Service (DoS) attacks.

The solution is simple to implement this verification server-side.

4.3.4 User Suggestions. Most user suggestions will be implemented. The priority is what can be done faster with greater returns in overall users satisfaction.

REFERENCES

[1] Anthonyjgrove. 2017. A react Google Login component. https://github.com/anthonyjgrove/react-google-login

[2] Stephen D Brown. 2007. *Fundamentals of digital logic with Verilog design.* Tata McGraw-Hill Education.

[3] Michael D. Ciletti. 2010. *Advanced Digital Design with the Verilog HDL* (2nd ed.). Prentice Hall Press, USA.

[4] Daniel C. Hyde. 1998. Using Verilog HDL to Teach Computer Architecture Concepts. In *Proceedings of the 1998 Workshop on Computer Architecture Education (WCAE '98).* Association for Computing Machinery, New York, NY, USA, 10–es. https://doi.org/10.1145/1275182.1275192

[5] Nathan Kozyra. 2015. *Mastering Go web services : program and deploy fast, scalable web services and create high-performance RESTful APIs using Go.* Packt Publishing, Birmingham, UK.

[6] Ícaro Lima and Elmar Melcher. 2018. Um Simulador Didático para o Ensino de SystemVerilog. In *Workshop sobre Educação em Arquitetura de Computadores.*

[7] Ícaro Lima, Elmar Melcher, and Joseana Fechine. 2021. Remote FPGA Lab for Distance Learning. In *sdfsdfs.* 28–31.

[8] Manuel Martínez-Almeida. 2014. Gin Web Framework. https://github.com/gin-gonic/gin

[9] Microsoft. 2016. Monaco Editor. https://microsoft.github.io/monaco-editor/

[10] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference.*

[11] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* 301–312.