The ECE 270 Verilog Simulator

Niraj Manikantadas Menon

ECE 49600 – 3 credit hours

Advised by Dr. Rick Kennell

Spring 2020

Purdue University

Author Note

Correspondence regarding this article must be addressed to Niraj Manikantadas Menon, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907.

University email: menon18@purdue.edu

Personal email: nirajmmenon@gmail.com

Starting this semester, the simulator has been made available to any Purdue student or faculty interested in trying out our FPGA-with-breakout-board-based Verilog simulator. One may create an account at https://verilog.ecn.purdue.edu/.

Project Description

This project was intended to extend the existing ECE 270 simulator developed in the previous semester, to introduce more secure access methods, improve upon the coding experience of the site, and publish the first version of the full documentation explaining the simulator internals.

Abstract

This document is intended to serve two purposes – to detail the changes that took place on the simulator during this semester, including the cluster-based simulation method whose development was accelerated by the COVID-19 crisis, and to provide a full-coverage documentation for the simulator detailing codebase functionality by file, overall simulator operation flow, inter-process communication between the various tools used, and the technical workings of the cluster. This document assumes that you, the reader, have used the simulator extensively as part of ECE 270, are interested in its internal workings, and has a fair amount of experience with UNIX shells, front-end web design, and/or JavaScript used client and/or server side.

The update involved adding numerous fixes, patches and new content to the web interface and the supporting server, as well as the inclusion of new features like cluster-based simulation and terminal communication. Some of these changes were planned to be deployed next semester, but due to the COVID-19 pandemic that forced the world, and subsequently, the Purdue campus, to move online, these changes were fast-tracked to make the simulator more accessible, realistic, and capable of handling large numbers of students simultaneously.

The full simulator documentation given in this document describes the top-down specification of the codebase used to implement the back-end of the simulator, which includes the web server instances that serve the initial HTML to students, the load-balancing code to redirect student simulations to our machine cluster in the EE 65 and 69 labs, and the code that runs on the cluster machines that will handle the Verilog simulations and send their outputs and/or errors back to the user.

This document is also intended to fulfill the author's ECE 49600 end-of-semester final report requirement for the semester of Spring 2020, while receiving valuable advice from my instructor, Rick.

Keywords: Yosys, CVC, node.js, cluster, simulation, Verilog, FPGA

The ECE 270 Verilog Simulator – Part I

The simulator was conceptualized in January 2019 as a solution to the problem of students being unable to test their Verilog-based hardware designs without being forced to attend open-lab sessions for unnecessarily long periods of time, of which most of that time was spent simply waiting to get on a computer to be able to use the development board. With this development board simulator, students' time in office hours was substantially reduced, and their reclaimed time was more effectively spent on learning the crucial concepts of the course.

Given Verilog code from a student, the simulator forwards it to the host server verilog.ecn, which has been reconfigured to redirect the code to the cluster of 76 machines situated in the EE 65 + 69 labs, aimed at distributing the now huge load of synthesis effort so that students don't see delays or hang-ups in the simulator. When a machine in that cluster picks up the code, checks for syntax errors, synthesizes it into a structurally formatted Verilog file, which is then simulated, its inputs derived from the pushbuttons on the student's webpage, and its outputs redirected to the LEDs and seven-segment displays on said webpage.

On the next few pages, the numerous changes and updates that were made throughout the semester will be discussed, as well as detailed statistics on simulator usage and how usage spiked during lab deadlines, exam periods, and the comparison before/after the shift to online classes and its effect on usage.

Changes to the simulator for the Spring 2020 semester

User account management system + login/signup/password reset portal

During the winter break after Fall 2019, the author decided to move away from basic auth as an outdated login method, and instead use session-based authentication for the simulator, which gave users the ability to log out and register, and himself the ability to revoke session IDs easily in the event that a session ID was being used to log in and run malicious code on the simulator.

The author designed a front-end portal page that visitors would be redirected to if they were visiting the site for the first time or were logged out users. Located at https://verilog.ecn.purdue.edu/portal, the page would allow users to log in as usual with their username and password, or non-users to sign up for an account with their Purdue email. The latter involves a verification step where a numeric token sent to the student's email (after verifying that it is a Purdue email) has to be entered into the page to ensure that the student actually owns that email. After verification, students will have to enter a new password, and are then redirected to the simulator's main page at https://verilog.ecn.purdue.edu/.

In keeping with the author's conviction that a Dark Mode must be available to everyone to ease the strain on their eyes while coding at night, the portal also features a Dark Mode switch for those who may need it. Figure 1 on the next page features a screenshot of the portal page in dark mode.

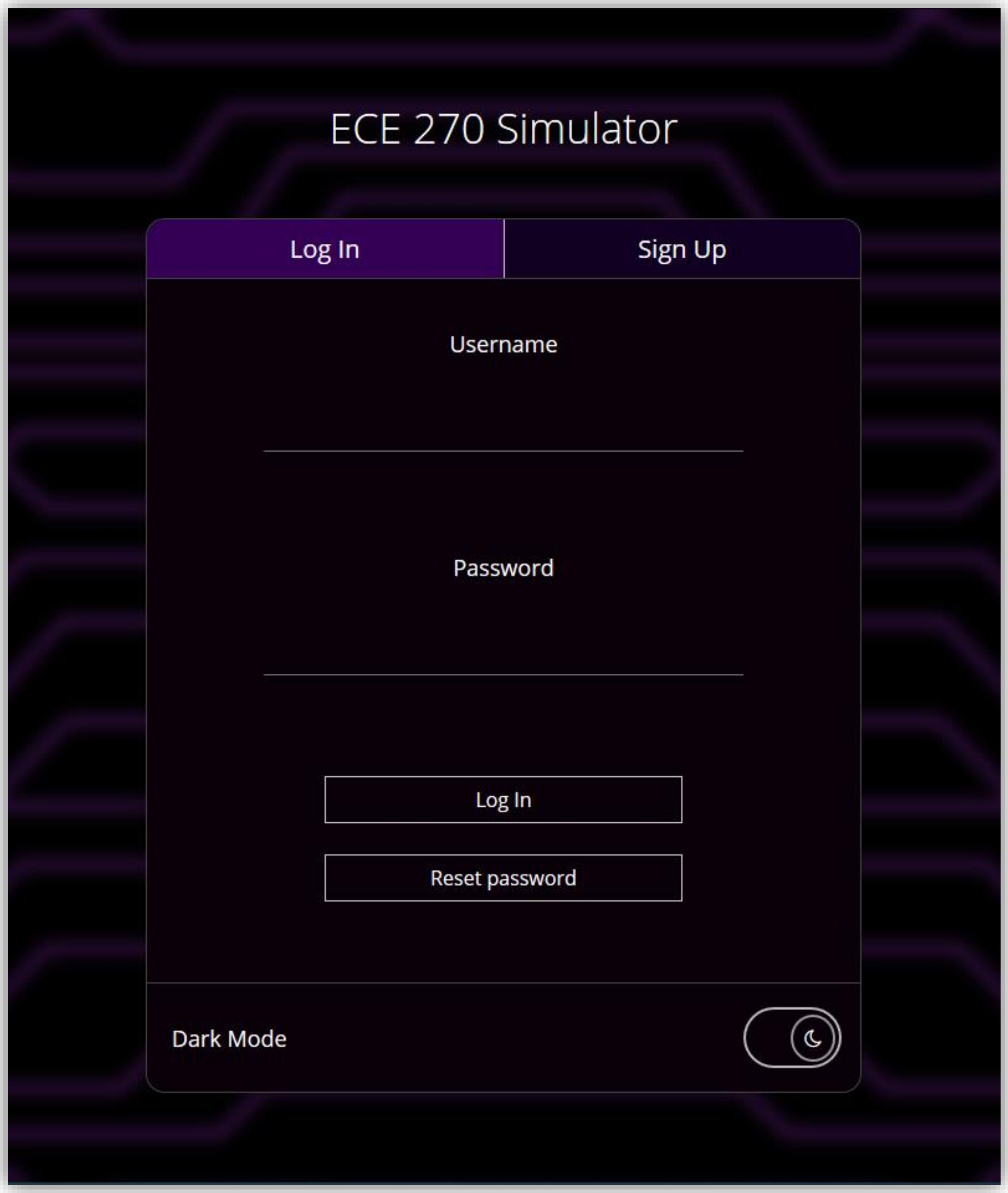


Figure 1: Dark Mode view of the simulator portal page.

Tabbed editor

In the author’s drive to make the simulator more IDE-like, tabs were found to be an extremely helpful feature that would take away the need to keep different code segments in separate files on the user’s computer. It created unnecessary clutter on students’ computers, and so a method of retaining multiple code segments on the simulator without having to copy it back in every time was a much-needed feature.

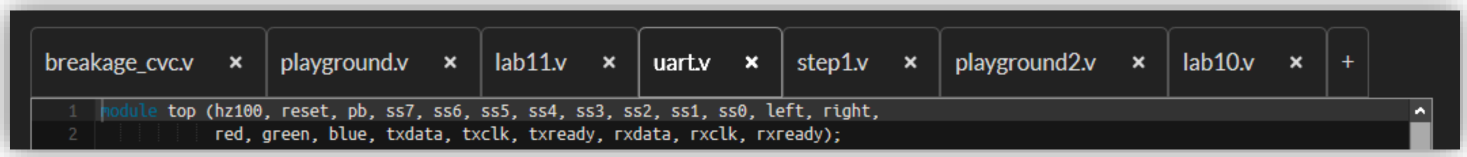
The tab system was therefore written to satisfy this requirement. It was written quite rudimentarily, with basic functionality like creating, renaming, and deleting tabs.

To create a tab, the student would press the + button indicated below, or by pressing Ctrl+Shift+T while inside the editor (pressing outside the editor would reopen the browser’s last open tab, so these shortcuts must be pressed while **inside** the editor).

To view the contents of a tab, you can click it directly, or you can scroll to that tab by pressing Ctrl+` while inside the editor (the character ` is a tilde, commonly found above the Tab key on Windows keyboards).

To temporarily close a tab, the student can press the “X” button associated with the tab, or by pressing Ctrl+Alt+W inside the editor while that tab is active.

To permanently remove a tab, the student can close the tab as specified above, then remove the tab entirely from the browser by using the file manager to remove the tab by name.



```
module top (hz100, reset, pb, ss7, ss6, ss5, ss4, ss3, ss2, ss1, ss0, left, right,
            red, green, blue, txdata, txclk, txready, rxdata, rxclk, rxready);
```

Figure 2: View of a tabbed editor, with a screenshot of the author’s tabs.

SystemVerilog support

The Yosys synthesis tool used to generate structural Verilog from a given student’s design, and the CVC simulation tool, included limited support for certain SystemVerilog constructs, such as:

- `always_ff -` Models a flip-flop with at least one clock edge, and at least one event run by the occurrence of that clock edge.
- `always_comb -` Models combinational logic as an always block instead of using the assign keyword to do so. The advantage comes when the logic is more easily written out as a case statement as opposed to a ternary operator.

- `always_latch` - Models a latch that can trigger events based on transitions in input signals, as long as conditions are met with those inputs.
- The `logic` keyword - Replaces the reg keyword, which caused confusion regarding how it would fit into hardware. While the keyword was not introduced this semester, it is assuredly a change that will be made if students make the switch to learning SystemVerilog.

However, the simulation tool, CVC, did not support the use of `typedef enum`, a very useful keyword helpful in generating state machine constructs and named states, as well as creating new data types. Since this was considered a game-changer, it was decided that SystemVerilog would not be introduced until support for this keyword came out.

For this semester, however, the `always` constructs were used to teach students the differences between the different sequential devices like flip-flops (ff) and (latch)es, and combinational logic (comb).

Admin control panel

Primarily intended for TAs, a rudimentary admin page was added at the /admin route which allows course staff to access error and simulation data for students. Currently the page only provides information regarding the student simulator account, such as the number of simulation and demo requests, and errors, as well as data corresponding to each error. It also provides insight on whether simulations are being handled correctly, whether logins to the site do not appear to come from unexpected locations, as well as the total number of simulations and access logs in the simulator's lifetime (which was reset to zero this semester). As of May 10th, 2020, the simulator has logged 340,816 total simulations, and 28,545 visits to the page.

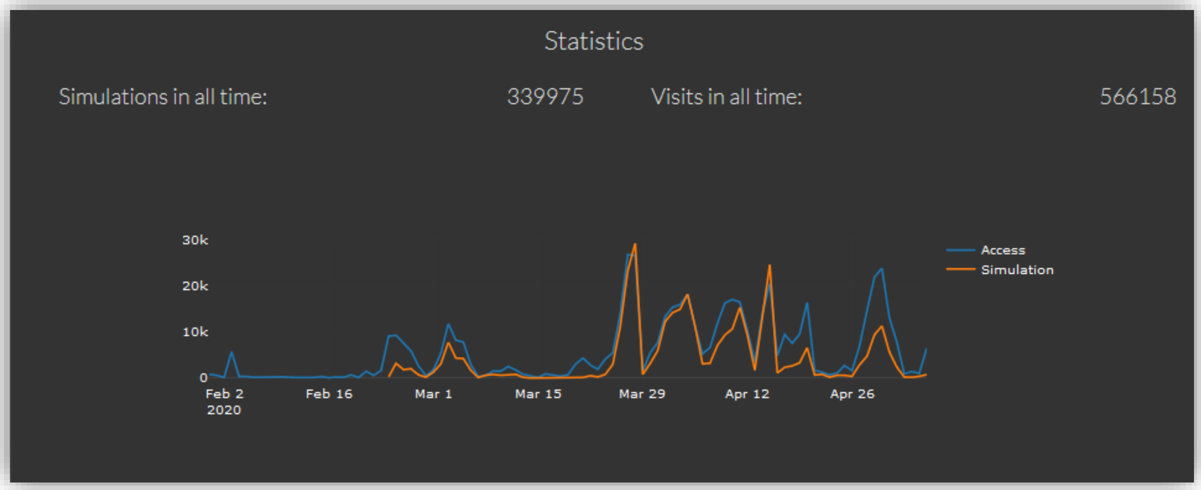
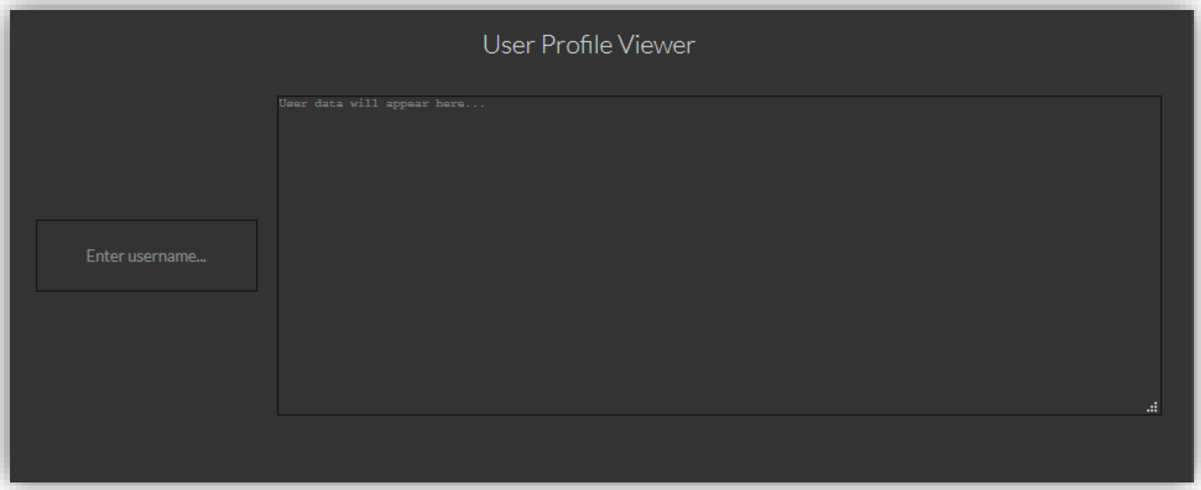


Figure 3: Screenshots of the User Profile Viewer (below) and Statistics views on the admin page.



Cluster-based simulation

The simulator's main server, hosted on an 8-core CPU, 64 GB RAM machine, handled simulations extremely well for students in the Fall 2019 semester, when load was extremely light, with a rough maximum of 10 students simulating their code simultaneously. However, the load quadrupled due to the more-than-doubled number of students (413 for the Spring 2020 semester) and the COVID-19 crisis forcing courses online. It was found very quickly that upon approaching the deadlines for assignments, a large number of students would use the simulator, all together, all at once, and these simulations would very quickly max out the CPU cores on the machine. The ceiling of simulations hit 40 before the server started lagging significantly, causing students to worry about the timing accuracy of their code, which at that point was intended to model counters, shift registers, and other sequential Verilog designs that required timing-accurate behavior for verification.

The simulator was then quickly redesigned to support a cluster-based model of simulation, which will be explained in the "Cluster Operation and Communication" section of this document.

Project results for this semester for ECE 49600

In addition to the aforementioned new features conceptualized, developed, and implemented during this semester, the author has also managed to achieve the following from last semester:

- There is now basic timing violation support in the simulator, implemented by delaying the data edge beyond the clock edge by a millisecond. This ensures that the data edge must be set well beyond the clock's setup region (which is typically a few nanoseconds).
- The maintainability of the simulator has been improved such that the server-side code is now separated by functionality, providing necessary abstractions to make the purpose of each module more understandable by keeping their code separate.

The ECE 270 Verilog Simulator – Part II

This part of the document is intended to serve as a whitepaper of sorts for the simulator. Such documentation was sorely needed to be able to explain the simulator to another technically proficient person. Since two stable versions of the simulator have been developed (stability being determined by the large numbers of students who used it for ECE 270 reliably without too many server crashes) it is best that the documentation for the simulator be finalized now.

The next few pages will explain, in painstaking detail,

1. The simulator codebase, with an analysis of the different components.
2. The set of operations performed on Verilog code received from a student as a flowchart.
3. The method of inter-process communication between the actual Verilog tools being used to synthesize and simulate code, and the node.js server handling the numerous connections from each student.
4. The operation of the cluster of machines being used to simulate student code, and how they interact with the main server using Redis.

The Simulator Codebase

To be thorough, this section will include:

- A full-page summarizing flowchart of the simulator codebase as operations are performed, starting from the top-level JavaScript code that instantiates the other files, indicated in Figure 4.
- An extensive description of each file in the codebase:
    - Main server JavaScript modules
        - `cluster.js`
        - `server.js`
        - `loadbalancer.js`
        - `admin.js`
        - `user.js`
    - Per-cluster node JavaScript modules
        - `websocket.js`
        - `normal_simulate.js / uart_simulate.js`
    - Client-side JavaScript for the webpage served to users
        - `simulator_backend.js`

Codebase layout flowchart

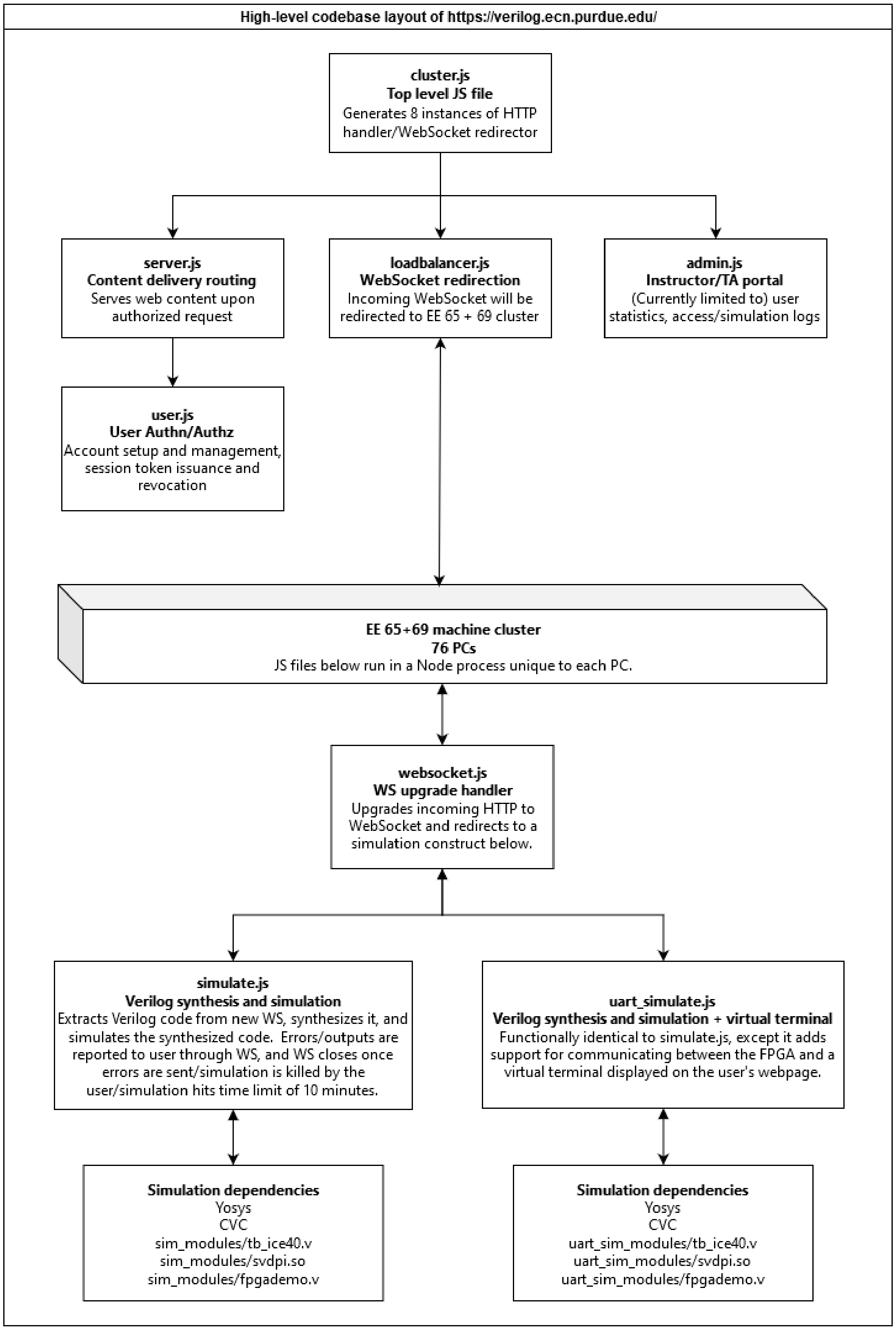

High-level codebase layout of https://verilog.ecn.purdue.edu/
cluster.js
Top level JS file
Generates 8 instances of HTTP handler/WebSocket redirector
server.js
Content delivery routing
Serves web content upon authorized request
loadbalancer.js
WebSocket redirection
Incoming WebSocket will be redirected to EE 65 + 69 cluster
admin.js
Instructor/TA portal
(Currently limited to) user statistics, access/simulation logs
user.js
User Authn/Authz
Account setup and management, session token issuance and revocation
EE 65+69 machine cluster
76 PCs
JS files below run in a Node process unique to each PC.
websocket.js
WS upgrade handler
Upgrades incoming HTTP to WebSocket and redirects to a simulation construct below.
simulate.js
Verilog synthesis and simulation
Extracts Verilog code from new WS, synthesizes it, and simulates the synthesized code. Errors/outputs are reported to user through WS, and WS closes once errors are sent/simulation is killed by the user/simulation hits time limit of 10 minutes.
uart_simulate.js
Verilog synthesis and simulation + virtual terminal
Functionally identical to simulate.js, except it adds support for communicating between the FPGA and a virtual terminal displayed on the user's webpage.
Simulation dependencies
Yosys
CVC
sim_modules/tb_ice40.v
sim_modules/svdpi.so
sim_modules/fpgademo.v
Simulation dependencies
Yosys
CVC
uart_sim_modules/tb_ice40.v
uart_sim_modules/svdpi.so
uart_sim_modules/fpgademo.v

Figure 4: Codebase layout of the ECE 270 Verilog Simulator

Codebase main server components

The simulator runs as a node.js process that starts with cluster.js, which in turn instantiates the other necessary JavaScript modules in the other files to provide necessary functionality.

cluster.js – Top level JavaScript module

Named after its purpose of starting a cluster of 8 instances of the simulator server on the 8 cores of the verilog.ecn.purdue.edu central machine, cluster.js is essentially the glue that connects the different modules together and provides functionality for the other modules to use. It performs the following functions.

- **Access/simulation count:** With every new line being added to the access or simulation logs, cluster.js will loop through the connected WebSocket connections, identify the ones that originated from the `/admin` page, and send a message with the updated access and/or simulation counts to be displayed on their webpages.
- **Loads server.js, loadbalancer.js and admin.js for adding necessary functionality:**
  - server.js will provide the underlying server object that will handle the raw HTTPS requests received by the node process at port 443. It yields the Redis `client` (used for writing data to a Redis DB), `sessionHandler` (used to authenticate any HTTPS request by checking credentials against the Redis DB) and `redisGetAsync` (used for reading data from a Redis DB) objects.
  - loadbalancer.js will provide a LoadBalancer module whose function is fairly straightforward – after being initialized, the module will forward WebSocket upgrade requests to one of the machines in the cluster using a round-robin method (i.e. to the computer after the last one that had received a WebSocket connection).
  - admin.js is a WebSocket handling module that immediately sends access and simulation data to be displayed at verilog.ecn.purdue.edu/admin when the connection between an authorized user who opened that page and the server is established. It sends data as it gets written to the access/simulation log files to all the users connected to the admin WebSocket module.
- **Instantiates HTTPS and HTTP redirect-to-HTTPS servers:** Even though server.js does the job of handling the HTTPS requests, there is no point unless it was used to configure an actual server bound to ports 443 and 80. It starts the HTTPS server with the certificates located in the LetsEncrypt default directory. As a result, when a HTTP request arrives at port 443, it will be received at cluster.js, where it is forwarded to server.js which will respond as needed as long as the request is authorized, if

necessary. The port 80 server will simply replace 'http://' with 'https://' in the incoming request URL and send it back as part of a 301-permanent redirect.

- **WebSocket routing:** Beyond just passing WebSockets to the LoadBalancer module, cluster.js also adds some extra WS routing capabilities. To be able to load-test the simulator, cluster.js also contains a route which does not require a session ID, but a certain query string to be specified in the URL. The query string is essentially a very long password that would allow authorized users (such as course staff) to test the reliability of the simulator by starting numerous simulations by connecting to this particular WebSocket route.
  There is also a 'banned' route to which all unauthorized WebSocket upgrade requests are forwarded, which simply upgrade the request to a WebSocket, send a message indicating that the WebSocket connection was not allowed, then close the connection.
  For regular WebSockets from authorized users, the route specified in the WebSocket URL determines whether the code includes the UART ports or not. If the WebSocket is connected to `wss://verilog.ecn.purdue.edu/`, it starts a normal/non-UART simulation, and when connected to `wss://verilog.ecn.purdue.edu/uart`, it starts a UART simulation.

server.js – Resource request handler with authorization methods

For those familiar with the node.js Express web application framework, server.js simply yields an Express app, as well as Redis communication and authorization interfaces (methods that will retrieve/add/modify/delete information in the local Redis database). An Express app is an interface that handles HTTP requests by the route specified in the URL by first authorizing the request if required (by checking if the username and session ID provided match and are valid) then returns the resource specified by the URL. The following are routes handled by server.js.

- **Content Delivery Routes**
  - `/assets` – Contains the core CSS and JS files and frameworks needed to style and provide functionality to the HTML pages in the simulator.
  - `/md` – Contains Markdown files used in the simulator handbook (located at `/help`)
- **All user routes at `/user/*` –** Functionality from user.js will be added to the Express app in server.js for the `/user/*` routes. Will be discussed in the user.js description below.

- **The Mind Palace at `/mindpalace`** – a cheeky reference to the mind palace memory technique used by Sherlock Holmes, which if you did not know, actually originated in Ancient Greece, and was originally called the method of loci.
  The `/mindpalace` route is simply a testing environment used by the author to try out web content and frameworks before integrating them into the simulator. This contains non-sensitive content, and cannot be browsed to find possible loopholes, so it does not require any prior authorization to access. This makes it very easy to demo new content even to users who may not have an account.
- **UART simulator at `/uart`** – This is different from the `/uart` WebSocket route in that this `/uart` route is handled when the request is a fully HTTP one. Since this particular page allowed simulations, it was not placed in mindpalace where it could be accessed and abused with no authorization, so this route was made to test the UART simulation process while being authorized.
- **The login page at `/portal`** – This is where an unauthorized user is directed when attempting to access a protected resources, like the main page at /, or `/uart`.
- **The main page at / -** When a user has successfully authenticated at `/portal`, they will be redirected to this route. This will load the main page of the simulator, where students will enter Verilog code to be synthesized and subsequently simulated.
- **The simulator handbook/help page at `/help`** – This was introduced in the Spring 2020 semester to provide some documentation aimed at helping students use the full capability of the embedded code editor, shortcuts, and file management. When only /help is requested, it returns the index.html located in the help folder. When /help/* (where * is any other resource file in the help folder) is specified, the resource "*" will be returned.
- **The administrator page at `/admin`** – A one-page application that gives statistics about the simulator usage, as well as its users. Its access is only restricted to course staff, and any attempt to log in by a student will be logged and responded to with a warning that their attempt has been logged and repeated attempts will result in an IP-based ban.
- All requests to the server are parsed for IP (from which the city and country is derived), type of request (GET/POST), username if any, requested resource, and the User-Agent string. All this information is logged in `simulator/logging/access.log`.

server.js also initializes three objects to be used by cluster.js:

- A `client` object, which provides the methods needed to interact with the local Redis database on the server, which contains all user data sorted by usernames, and simulation statistics for each cluster node.
- A `sessionHandler` object which is used by cluster.js to identify authorized WebSocket upgrade requests before forwarding them to a cluster node for handling the upgrade.
- An asynchronous `redisGetAsync` function that will retrieve data associated with a specified key from the Redis database.

loadbalancer.js – The Load Balancer Redirection module

The function of the LoadBalancer object (returned by this module when it is loaded by cluster.js) is simply to accept a HTTP request object containing a WebSocket `“Connection: Upgrade”` header, select a node in the cluster of machines located in the EE 65 and 69 labs in the EE building at Purdue based on a non-biased round-robin method (routes a simulation to the first machine on initialization, and the next one will go to the second machine, then the third, and so on until it reaches the last machine in the list and loops back around, regardless of machine load).

Since the server application runs as eight instances, one might find that the LoadBalancer module will not work correctly because there are eight instances of itself, and therefore eight counters that will not necessarily point to the same next server. This problem is resolved by using a single counter in the Redis application, called `node:index`, which will contain the index of the last server to which a WebSocket request was routed.

Since this is relatively new, there are still bugs involving routing to a cluster node that might be down, causing simulations to fail with no notice, or if one computer becomes way too overloaded with simulations by coincidence. In such events, the code will simply move to the next computer node, but will eventually be extended to check the process and usage statistics for each node to make a more intelligent decision on how to pick the least busiest node to forward the WebSocket simulation to.

admin.js – WebSocket handler to serve content for `/admin`

The admin.js file implements a WebSocket handler that receives a just-upgraded-from-HTTP-to-WebSocket connection, with the URL set as `/admin/admin_ws`, and sends simulator usage and user data upon request from the course staff member viewing the admin page.

The data available to course staff members through this module currently consists of the following.

- **Number of access/simulation attempts**: When the connected user requests access/simulation counts using the term "`req_count`", admin.js will return the line count of the access log file located on the main server, and the simulation log file on the author's NFS mount used by the cluster nodes, respectively.
- **User data:** In order to access (most importantly) the error logs and count, as well as other data associated with a user account, the connected user must send a JSON string of the format `{username_data: "username"}` from which admin.js will extract the username, query the Redis database for the data associated with the username, and send back the raw JSON stored under that username in Redis.

Since usage statistics and student data stored on the server is not very sensitive data, it was decided that that would be the only information available through a web portal for course staff, hence the rudimentary nature of this file.

user.js – User Management System

This file provides user-specific functionality to the aforementioned Express app in server.js, specifically creating user accounts and initializing them in the Redis DB, issuing session tokens to existing users upon login, revoking session tokens for existing users upon logout or timed expiry, and password change handlers.

- **`/signup`** – GET and POST methods to allow a visitor to sign up
  - **GET method –** User first sends an email in the query string from the /portal page, so the URL looks like:
    - `https://verilog.ecn.purdue.edu/signup?email=pete@purdue.edu`
  - Upon verification of the email, a verification token is emailed by the server to the specified email, and the token is stored in Redis along with the specified email.
  - The user receives the token in his email and types it into the /portal page and sends it. The URL that arrives at the server should now look like:

        - `https://verilog.ecn.purdue.edu/signup?email=pete@purdue.edu&token=123456`
    - If the token and email pair matches that which is stored on Redis, the user has successfully proved that he owns the email account specified.
    - **POST method –** This is called when the user is prompted to enter a new password after verifying the token. Since students' passwords should not be made visible in the access logs (which they would be if placed in the query string), it will be sent in the body of a POST request instead, which does not appear in access logs. To ensure a student cannot just change someone else's password with this method, the token-email pair must be sent again in the query string to ensure that it is still the same user, now setting up their password. As a result, the URL will look like this:
        - `https://verilog.ecn.purdue.edu/signup`
    - While the POST body will look like this:
        - `email=pete@purdue.edu&token=123456&password=AVeryLongSentenceIsAGoodPasswordITellYa2020!`
    - Assuming the email and token in the body still match the pair stored in Redis, the password for the user is hashed with a salt using the bcrypt module in node.js and stored into Redis in the data under their username. The email/token pair is deleted to prevent further modifications to the password, and a randomly generated UUID and username will be sent as part of a Set-Cookie header to provide authorization cookies - credentials for the user to use in all future interactions with the server, set to expire after 24 hours, requiring another login from the /portal page.
    - The reason the author did not use a POST method to handle everything while writing the initial code was due to debugging necessities, and it did not change ever since. In the event of a third full codebase rewrite, the author may consider merging these two methods into one POST method.
    - To avoid redundancy, the author actually also uses this method to handle password resets from the /portal page, since it also requires an email, subsequent token verification, and password entry.

- **`/login`** – POST method to set authorization cookies after authenticating user

- The user sends their username and password as part of a POST body from the /portal page on the Log In form, to https://verilog.ecn.purdue.edu/login.
- If the user exists and the provided password matched the stored password hash, recreate the session data for them if needed, which will reset the 24-hour expiry on the user's authorization cookies. Generate a unique UUID, store it under the user's session data on Redis, and send it using a Set-Cookie header in the HTTP response, which will contain a status message indicating successful login.
- If the user does not exist, send back a message indicating no such username.
- If the user's password hash did not match, send back a message indicating incorrect password.
- In future versions, the author plans to implement brute-force attack prevention techniques, like key stretching and/or fail2ban support for the logfiles.

- **`/passwd`** – POST method to change password (legacy, will be updated/removed later)
    - This function is intended to change the password if the user is already logged in. This was back from when the author had used Basic Auth to let users log in, but it was extremely buggy and did not allow session handling.
    - When a user submits a password change request from the simulator main page, it arrives with the user's username and new password in the POST body, and authorized session cookie containing the UUID associated with the user.
    - If the UUID is associated with that user, the password hash for the user is updated in Redis.
    - This method may also include brute-force prevention techniques later on, possibly using the node package `express-brute`.

Cluster-node-hosted JavaScript modules

When a WebSocket upgrade request is forwarded by the main server to the lab machine cluster, it arrives at a specific machine, upon which will be running a node.js process that has been started with the websocket.js file.

The beauty of this setup lies in how easily the code is replicated across all the machines as a result of the way Purdue's ECN department sets up user profiles on Linux machines.

When a Purdue student logs in to a Linux machine (including all the computers in the EE 65 + 69 labs, of which the 44 EE 65 computers were used for ECE 270 students, and 32 EE 69 computers were used for ECE 362 students, making a total of 76 machines), their user data, which is stored on a user profile server called shay, is loaded instantly through an NFS mount from the Linux machine to shay, authorized by the student's credentials. This also holds true for all remote-based logins as well, including SSH.

The author made use of this fact to write two JavaScript modules – one to act as a WebSocket server that would receive the forwarded WebSocket upgrade requests from the LoadBalancer module on verilog.ecn.purdue.edu, named **`websocket.js`**, and the other to handle the Verilog code processing and simulation, which is mostly identical to the former simulate.js module that was used for simulations on the main server, named **`normal_simulate.js / uart_simulate.js`**. The latter two files are identical except that they both use different files for simulating student code, as explained later.

While the functionality of the two modules has been very succinctly explained in a flowchart in the next section, a textual description of the modules are given as well, for a more verbose description.

websocket.js – WebSocket server

This module is a combination of WebSocket upgrade handler, connection multiplexer and process cleanup utility.

- **Connection handling –** The node process runs a regular HTTP server on port 4500, which is only accessible to machines in the same LAN. The server filters out connections by their origin – any connection originating from a machine other than verilog.ecn.purdue.edu, or ataraxia.ecn.purdue.edu (the dev server) will be dropped. If the connection satisfies the criteria, however, it will be routed to one of two simulation handlers – the non-UART one from normal_simulate.js, or the UART one from uart_simulate.js.

- **Hanging process watchdog –** If there are any CVC processes on the system that have hit 100% CPU usage, it is an indication that those processes have hit infinite loops, and will not end themselves, potentially causing the machine to crash if enough of them are allowed to run. Every half second, websocket.js will run through a process listing to find such processes and kill them with a SIGKILL signal. The SIGKILL ensures that the process dies – gracefully or not – which will indicate to the simulation handler JS code that that particular CVC process had gone bad, and the user should be notified.
- **Simulation statistic reporting –** websocket.js will update Redis on the main server with how many simulations it is currently handling through a UNIX domain socket, connecting from the machine to the server with the help of an SSH tunnel. As a result of outside access, Redis had to be secured with a long password that is stored on the server and in the author's profile data with secure permissions, and is read by the processes on both the server and cluster machines when they first start.

normal_simulate.js / uart_simulate.js – Simulation handler

These are the modules that the author genuinely believes to be the crux of the entire simulator. The idea of inter-process communication between the Verilog synthesis/simulation tools and the web application that these modules implement is how the main purpose of the simulator – to process Verilog code, synthesize and simulate it – is achieved.

The functionality of these modules is far better explained in flowchart form (in the simulator operation flowchart presented further down) than textually, since there is not much else to these modules than their intended functionality, as opposed to the websocket.js module discussed above, which, in contrast, has its fingers in a lot of pies, and therefore does require a textual description.

Client-side JavaScript – simulator_backend.js

To be fair to the numerous JavaScript frameworks and modules that actually provide a lot of the functionality of the main page of the simulator, simulator_backend.js (a misnomer from early days of development – a more accurate name would be frontend.js) is more like the glue that binds these frameworks and components together. The code in the file will be referred to as the "frontend JS".

The following features are implemented using simulator_backend.js:

- **Website control and customization –** To develop the simulator, the author had to design an intuitive and eye-pleasing interface to act as a virtual FPGA development environment, which meant that functionality and aesthetic were equally important. To that end, the frontend JS implements/handles (although not verbatim in the file) a key + shortcut handler, color scheme modifier, simulation info panel.

- **File management system –** The file system implemented in the browser is used to maintain a list of code segments, along with user-assigned filenames, and timestamps, in the browser's LocalStorage component, which allows a user to reopen the simulator tab at any point in time to continue working on what they were doing in their previous session. Typing into the editor fires a function that will immediately push changes to LocalStorage, so even if a crash occurs, a user could just pick up right where they left off after reopening the simulator.
- **Tab system –** As discussed in part 1 of this document, the editor also features tabs, which required maintenance on the number of tabs, the code segments associated with them, and add/create/delete/save functionality. The tab data is also stored in browser LocalStorage, with all information regarding the tabs for that session. Performing any tab operations will also immediately save changes.
- **Terminal view –** Also introduced in the Spring 2020 semester, the editor can now be hidden to make way for a terminal view, which allows for a simulated UART on the server-side to send characters to be displayed. This is meant to be analogous to attaching a serial communication program like `minicom` or `screen` to a UART serial port on the FPGA, programming the FPGA with a Verilog design to use the UART protocol to send and receive data through the TX/RX lines. On the simulator, however, the server fakes a "UART" for exchanging communications between the student design and the virtual terminal, so that students do not have to worry about implementing the actual UART protocol itself.
- **Panel system –** there are three panels which handle file management (mentioned above), UART settings and miscellaneous settings.
  - In the Settings panel:
    - **Dark Mode switch –** Crucial to a lot of users who prefer coding in the dark, the simulator offers a dark mode switch to alleviate the strain on their eyes in dark environments.
    - **Simulate-on-Save –** The habitual coder will always press Ctrl+S regularly to save their code. For users of the simulator, this can also be used to immediately simulate their code when they hit the shortcut.

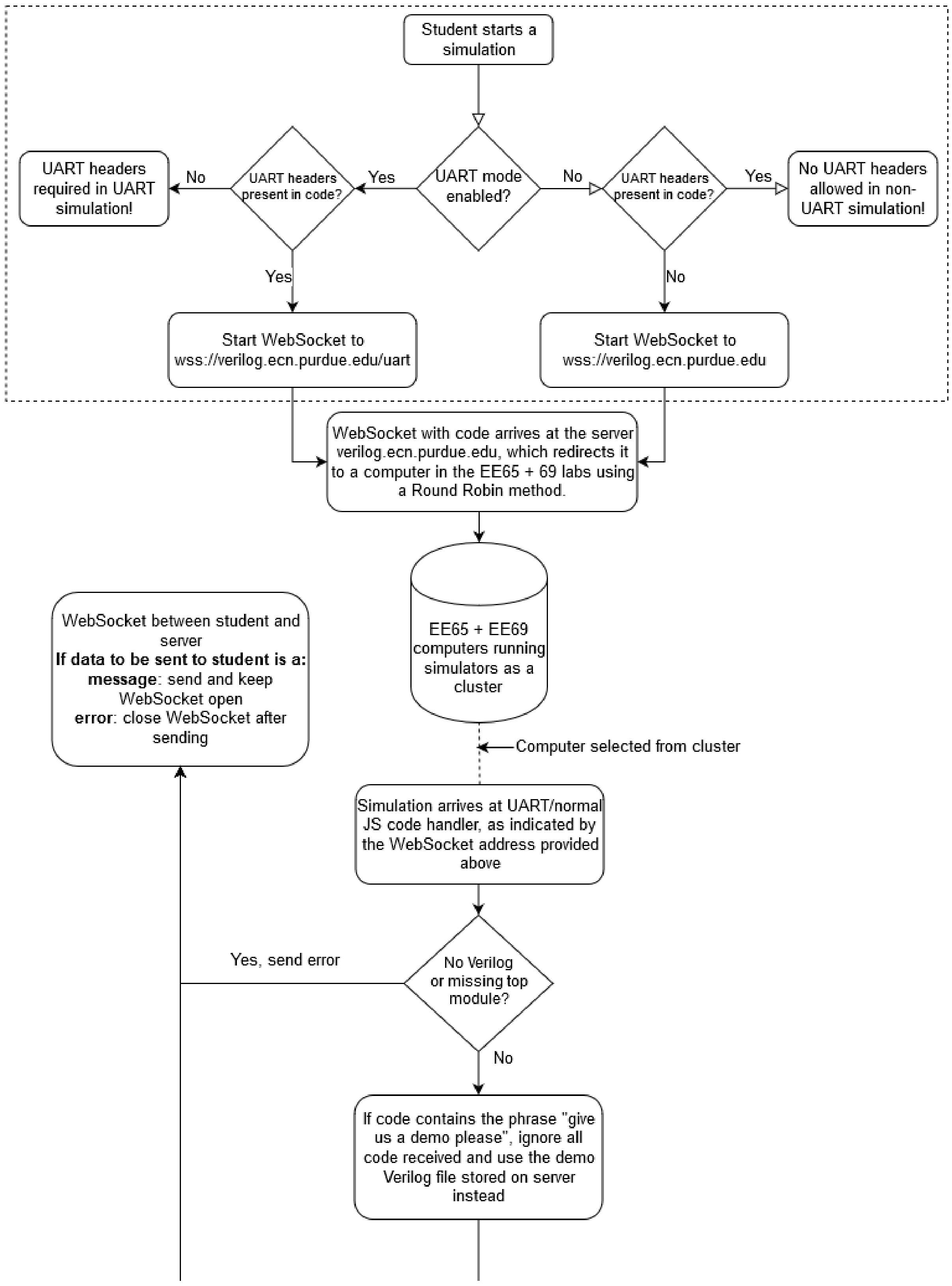
This takes away having to free a hand to use the mouse to click the Simulate button, and lets the user keep their hands on the keyboard.

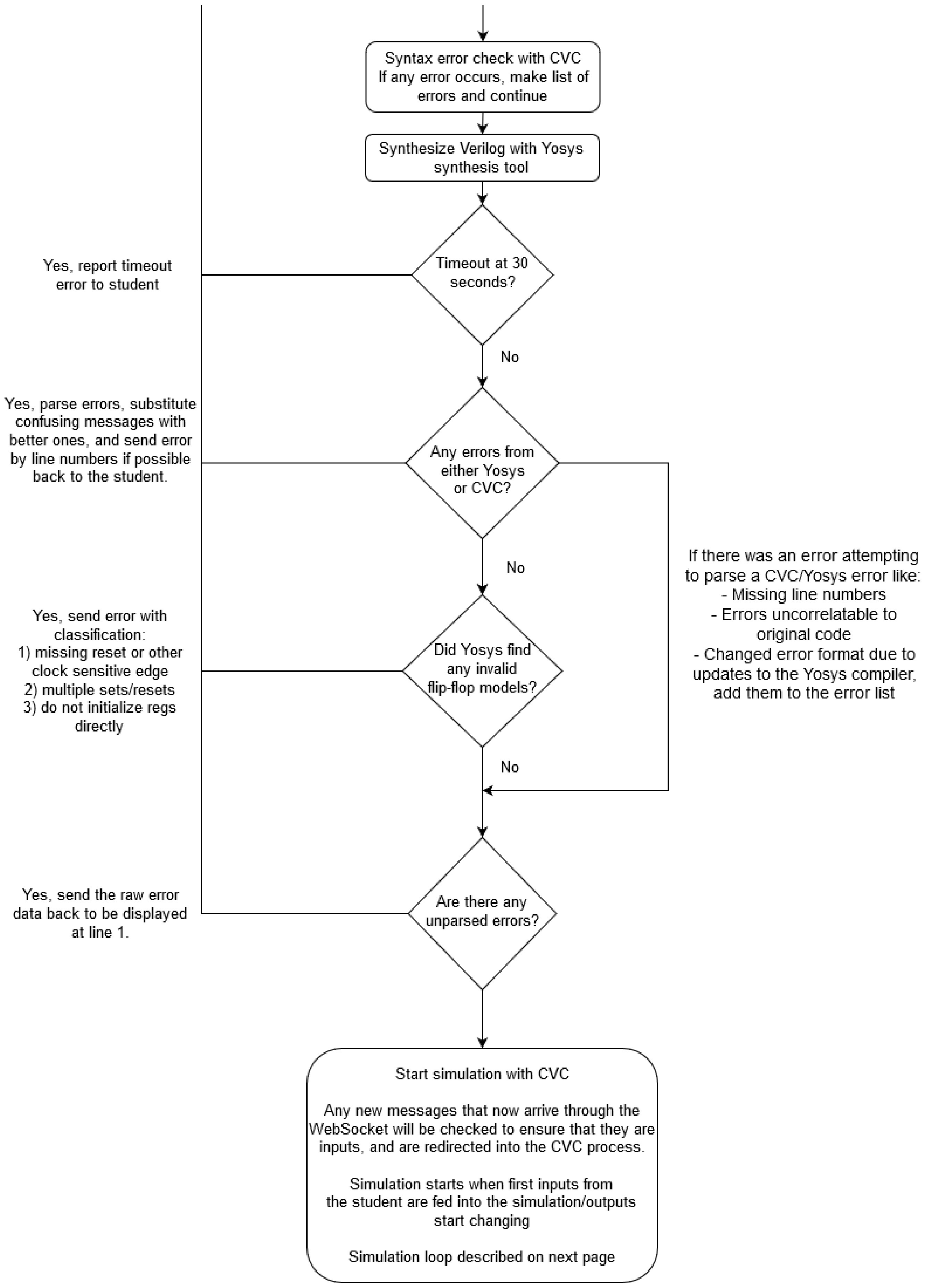
        - **Toggle Autocomplete –** When the author found and added an autocomplete feature for Verilog for the editor, it was found (even by quite a few other students) that the autocomplete feature was a bit of a hindrance while typing constants or similar variable names. To that end, the author added a switch for users wishing to turn this off.
        - **Password change –** If a user simply wishes to change their password while logged in, they can simply enter it in the box alongside the panel and hit Enter. A visibility toggler is also in place to ensure they do not type the wrong password.
        - **Autosave interval –** The code also gets autosaved every 2 minutes, so if users wish to change that time period, they can enter a new value in milliseconds in the box on the right.
        - **Evaluation board color theme –** Adds additional theming for the virtual FPGA board.
    - In the UART panel:
        - **Enable UART:** Enables UART mode to ensure that the code is sent to the UART simulation module on the server-end, and not the normal one.
        - **Auto-switch to Terminal –** When a simulation starts, the user will automatically view the terminal. Useful for when the user immediately wants to see the output on the terminal at the beginning of a simulation.
- **Simulator Handbook –** For users wishing to see a usage manual for the simulator, including a full changelog, tips and tricks, quick start (and this documentation in the future) etc. they can click the Help button at the top of the page.
- **Simulation handler –** The core part of the frontend JS, the simulation handler works within one of two contexts:
    - **Student code simulation –** When a student enters Verilog code in the editor and hits Simulate, the corresponding handler first makes some checks:
        - If the code contains the Unicode apostrophe, it is an indication that the code has been copied from the instructor’s lecture notes without being typed out correctly. Issue a warning and do not continue.

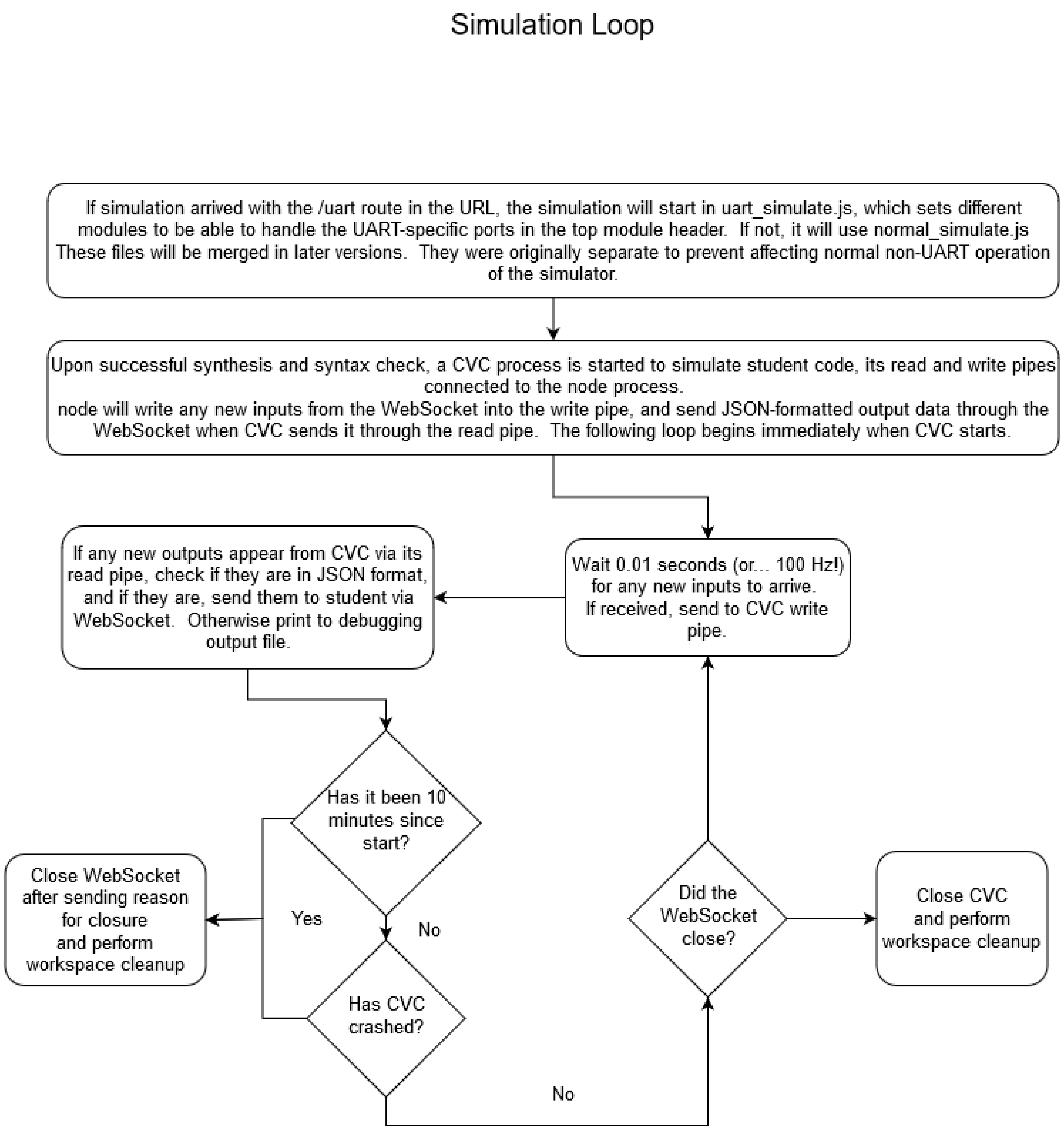
        - If the code contains references to the simulated UART header ports, but UART mode for the simulator was not enabled, or vice-versa, issue a warning. (This will be removed in future versions since the simulated UART does not require that the ports be actually used.)
        - Check that the code does contain a top module before simulation. If there are occurrences of a commented out top module, issue a warning and stop.
        - If a simulation was already running, stop that simulation.
    - If these checks are satisfied:
        - Start a WebSocket back to the server, specifying UART/non-UART in the URL string.
        - Change the status field to Connecting.
        - Specify the following handlers:
            - `ws.onopen`: once the WebSocket has been successfully routed to a cluster machine on the server end, the server will send back a Sec-WebSocket-Accept header, which tells the frontend JS that the WebSocket is now open.
            - `ws.onmessage`: After receiving the initial messages indicating successful connection and synthesis, any future message can be either one of two things – errors, which will be parsed and displayed on the editor or used to stop the simulation, or outputs, which will be parsed and displayed on the virtual FPGA.
              If UART was turned on, along with the Auto-Switch to Terminal option, the view will switch from the editor to the virtual terminal.
    - **Demo handler –** Works exactly the same way as the simulation handler, but it does not perform any checks, and sends a string "give me a demo please" in place of the code.
- **Stop/Reset simulation –** Stop will cause the FPGA to freeze, while Reset will clear the outputs on the FPGA. Both will kill the simulation and close the WebSocket.

Simulator Operation Flowchart

The part circled with a dotted line indicates client-side operation, starting from the moment the student presses the Simulate button. After that, the remaining operations take place on the server.

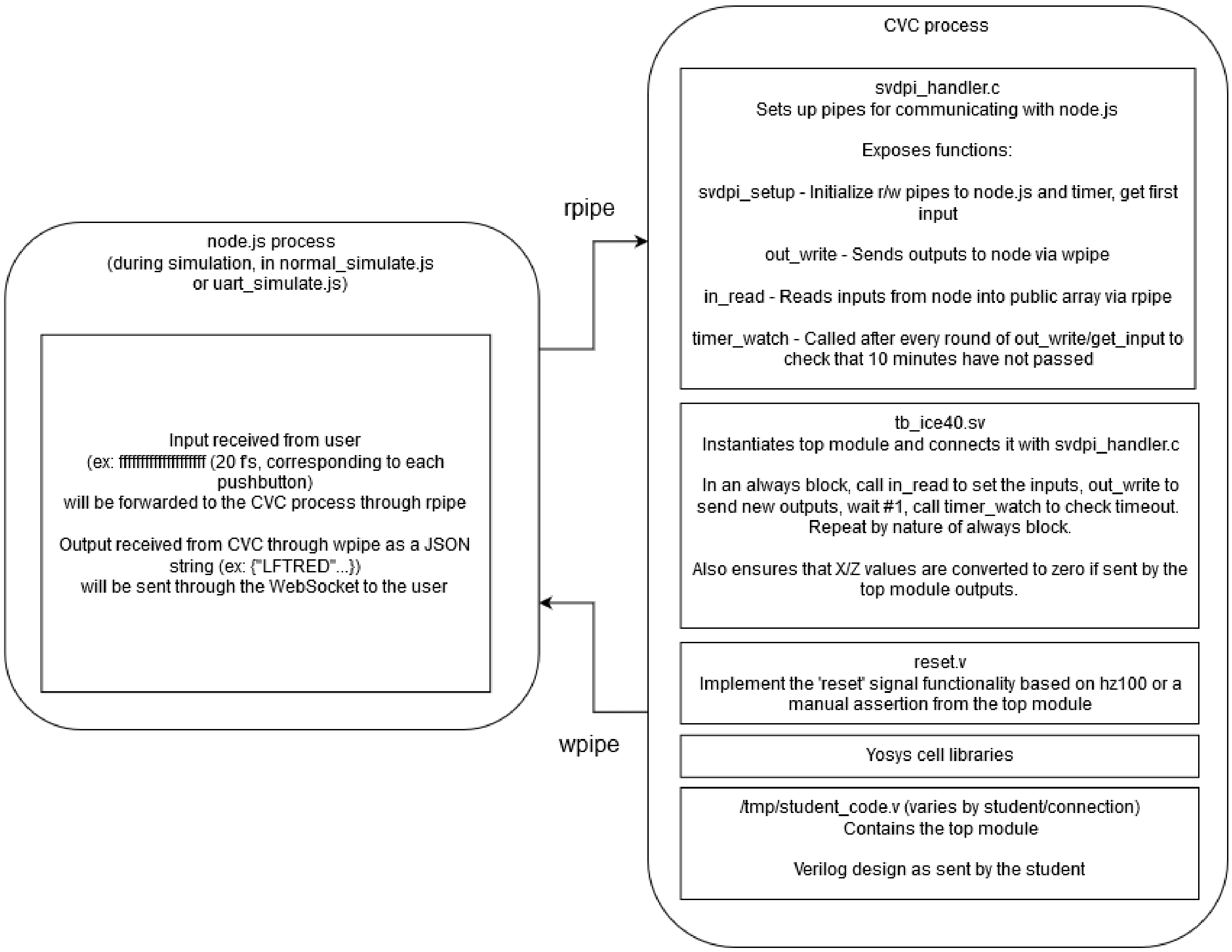

Student starts a simulation
UART mode enabled?
Yes
No
UART headers present in code?
No
UART headers required in UART simulation!
Yes
Start WebSocket to wss://verilog.ecn.purdue.edu/uart
UART headers present in code?
Yes
No UART headers allowed in non-UART simulation!
No
Start WebSocket to wss://verilog.ecn.purdue.edu
WebSocket with code arrives at the server verilog.ecn.purdue.edu, which redirects it to a computer in the EE65 + 69 labs using a Round Robin method.
EE65 + EE69 computers running simulators as a cluster
Computer selected from cluster
Simulation arrives at UART/normal JS code handler, as indicated by the WebSocket address provided above
No Verilog or missing top module?
Yes, send error
No
WebSocket between student and server
If data to be sent to student is a:
message: send and keep WebSocket open
error: close WebSocket after sending
If code contains the phrase "give us a demo please", ignore all code received and use the demo Verilog file stored on server instead


Syntax error check with CVC
If any error occurs, make list of errors and continue
Synthesize Verilog with Yosys synthesis tool
Timeout at 30 seconds?
Yes, report timeout error to student
No
Any errors from either Yosys or CVC?
Yes, parse errors, substitute confusing messages with better ones, and send error by line numbers if possible back to the student.
If there was an error attempting to parse a CVC/Yosys error like:
- Missing line numbers
- Errors uncorrelatable to original code
- Changed error format due to updates to the Yosys compiler, add them to the error list
No
Did Yosys find any invalid flip-flop models?
Yes, send error with classification:
1) missing reset or other clock sensitive edge
2) multiple sets/resets
3) do not initialize regs directly
No
Are there any unparsed errors?
Yes, send the raw error data back to be displayed at line 1.
Start simulation with CVC
Any new messages that now arrive through the WebSocket will be checked to ensure that they are inputs, and are redirected into the CVC process.
Simulation starts when first inputs from the student are fed into the simulation/outputs start changing
Simulation loop described on next page


Simulation Loop
If simulation arrived with the /uart route in the URL, the simulation will start in uart_simulate.js, which sets different modules to be able to handle the UART-specific ports in the top module header. If not, it will use normal_simulate.js These files will be merged in later versions. They were originally separate to prevent affecting normal non-UART operation of the simulator.
Upon successful synthesis and syntax check, a CVC process is started to simulate student code, its read and write pipes connected to the node process. node will write any new inputs from the WebSocket into the write pipe, and send JSON-formatted output data through the WebSocket when CVC sends it through the read pipe. The following loop begins immediately when CVC starts.
Wait 0.01 seconds (or... 100 Hz!) for any new inputs to arrive. If received, send to CVC write pipe.
If any new outputs appear from CVC via its read pipe, check if they are in JSON format, and if they are, send them to student via WebSocket. Otherwise print to debugging output file.
Has it been 10 minutes since start?
Yes
No
Has CVC crashed?
Close WebSocket after sending reason for closure and perform workspace cleanup
No
Did the WebSocket close?
Close CVC and perform workspace cleanup

Inter-Process Communication between SystemVerilog DPI in CVC and node.js process

In order to exchange inputs and outputs with CVC, node.js sets up two pipes – a read pipe and a write pipe – connected to the standard input/output pipes of CVC, which is launched with the `svdpi_handler.c` object file (called `svdpi.so`), a middleman testbench (`tb_ice40.sv`), a reset functionality module (`reset.v`), Yosys cell libraries, and the student code.


CVC process
svdpi_handler.c
Sets up pipes for communicating with node.js
Exposes functions:
svdpi_setup - Initialize r/w pipes to node.js and timer, get first input
out_write - Sends outputs to node via wpipe
in_read - Reads inputs from node into public array via rpipe
timer_watch - Called after every round of out_write/get_input to check that 10 minutes have not passed
tb_ice40.sv
Instantiates top module and connects it with svdpi_handler.c
In an always block, call in_read to set the inputs, out_write to send new outputs, wait #1, call timer_watch to check timeout. Repeat by nature of always block.
Also ensures that X/Z values are converted to zero if sent by the top module outputs.
reset.v
Implement the 'reset' signal functionality based on hz100 or a manual assertion from the top module
Yosys cell libraries
/tmp/student_code.v (varies by student/connection)
Contains the top module
Verilog design as sent by the student
rpipe
wpipe
node.js process
(during simulation, in normal_simulate.js or uart_simulate.js)
Input received from user (ex: ffffffffffffffffffff (20 fs, corresponding to each pushbutton) will be forwarded to the CVC process through rpipe
Output received from CVC through wpipe as a JSON string (ex: {"LFTRED"...}) will be sent through the WebSocket to the user

CVC implements a feature called the Direct Programming Interface, a feature of SystemVerilog that allows C code to be written to drive CVC, and subsequently the I/O ports of the Verilog design being simulated. This neat trick is what allowed the author to write svdpi_handler.c to set up r/w pipes between CVC and node.js, allowing node.js to connect and to exchange I/O data with the student's Verilog design through the layers of abstraction noted above.

Cluster Operation and Communication

Although this has been discussed quite a bit, this section will explain the connectivity and communication processes between the node processes that run on the cluster nodes (the EE 65 + 69 machines), the main node process and the Redis server that runs on verilog.ecn.purdue.edu.

How does the main server forward simulations to the cluster nodes?

When the LoadBalancer module on the main server node process routes an HTTP connection (one containing the WebSocket upgrade headers) to the next server, it does so by resolving the hostname for that machine (which takes the form `ee6XlnxYY.ecn.purdue.edu`, where `X` is either `5` or `9`, `YY` will be `01..44` or `01..32` respectively.) to an IP address, to which the HTTP connection will be routed on port 4500 (which is open for all local connections). Once a machine receives the connection successfully, it will upgrade the WebSocket to indicate receipt, and process the code that gets sent thereafter.

How is code maintained and run across all the cluster nodes?

All the relevant files for the cluster nodes, including code for each cluster node (websocket.js, normal_simulate.js and uart_simulate.js), the simulation log, the password to access Redis on the main server, all sit on an NFS share on the machine shay.ecn.purdue.edu, which gets loaded upon logging in to any of the cluster node machines (the EE 65 + 69 machines) remotely or locally. The advantage of this is that code consistencies across the 76 machines is never a problem, so an update to the any file on the author's files on this NFS share would be replicated across the 76 machines.

To run the code simultaneously on all machines, websocket.js is started with node on every machine on the cluster in parallel with the help of the invaluable Ansible deployment tool on the main server. A playbook was designed to be run with Ansible that would do the following on each machine on the server:

- Create a temporary workspace directory where code would be processed.
- Set up crontabs to perform daily workspace cleanup and CVC process cleanup (for when the CVC processes do not appear to be using any CPU, but have been running longer than 10 minutes, which is when CVC is expected to quit).
- Create the UNIX domain socket if it did not exist, to allow access to Redis on the main server for the simulation handler to update its status on when needed.

- And finally, start the node process with websocket.js as the starter code, which in turn would load the *_simulate.js modules. Upon startup, websocket.js will connect to Redis through the UNIX domain socket established earlier.

Ansible connects to each machine via SSH and starts a shell to run the necessary commands to perform the aforementioned actions.

The node process on each cluster node will run on port 4500 to receive that connection. It is then validated by checking if the origin address is from either verilog.ecn.purdue.edu or ataraxia.ecn.purdue.edu, and if it contains certain keywords that could only have been sent from either aforementioned server.

How can the cluster node report how busy it is in terms of simulations?

While the current version of the main server does not necessarily use the number of simulations to inform the choice of cluster node, and instead uses a round-robin method of determining the next cluster node, this question is asked in order to lay the foundation for writing load-balancing code that would take into account the server load.

The purpose of the UNIX domain socket discussed earlier is to set up a secure tunnel back to the main server for the sole purpose of relaying information through the Redis server hosted there. A cluster node, upon receiving/ending a simulation connection, will update its total simulation count on that Redis server. As a result, for each cluster node, there is now a load statistic based on the number of connected simulations that could be used to inform future simulation routing.

Known Bug List

- Undo operations on the code editor are not separate by tab. If you perform an operation in one tab, then undo in another tab, the change from earlier will be redone in the open tab, and not the change intended for that tab.
- (More of a quirky Easter egg) Typing the line "give us a demo please" into the code editor, even if the user isn't starting a demo, will cause the demo program to be run instead of whatever was in the student code provided. This is a bug because the simulation JavaScript on the cluster will simply look for the phrase within code, but not equate the entire code to that string, as it should.
- The xterm.js module used for the terminal does not resize as expected when the code editor size is changed. This may cause text to go off screen, or prematurely move to the next line, in case of large amounts of text being printed on the terminal.
- CVC does not always simulate student code correctly, even if it is functionally and syntactically correct. The particularly odd thing about this situation is that the code works on the simulator, but not through the command line by running CVC directly. While the exact cause is unknown, the outdated version of CVC that is being used could do with an update that could fix these issues, or even a replacement with a more powerful and actively maintained project like Verilator, although that has its own drawbacks.
- If a cluster node goes down, the main server may not adjust accordingly to switch to a new server. Functionality will be added to move to the next available server if such a situation arises, possibly with a notification to the author of some sort that would indicate that a cluster node is down.
- Tabs do not get hidden on page refresh despite having been closed earlier, causing a large wave of tabs to get opened simultaneously resulting in missing tabs due to the unnecessarily long tab header. This should be fixed when the author rewrites the tab handler entirely, to behave in a way similar to tab systems in browsers and/or code editors.
- The simulator does not work reliably on Mac, which is a result of a lack of testing. Currently, there is no way of testing it due to lack of hardware, but it will be discussed at some later point.
- Improve editor autocomplete so that it gets less in the way and is more helpful for filling out longer keywords or defined variables.

- Better keyboard shortcuts and key handling so that one does not have to ever touch the mouse while working with the simulator, allowing for speedier coding, debugging and assignment completion. This also includes the case where, at times,
- Add support for different layouts (e.g. Dvorak) so that users with different keyboards may use the keys to manipulate the virtual FPGA display. This can be done easily by looking directly at the value of the key, and not the keycode (which has been deprecated at the time of writing this document).
- Switching between non-UART and UART was too much of a hindrance. It was found that the course can run just fine by leaving in the UART ports and just not use them.

Feature requests/improvements/ideas

Regression testing

- Incredibly crucial to the development of Verilog simulation tools is the application of regression testing, which is the practice of ensuring that new features do not get integrated at the cost of existing ones.
- Two regression testing modules should be added – one to ensure the proper manipulation of the webpage, and one to ensure that student code does not cause the simulator to crash with no warning.

Testbench-based verification module

- One of the biggest losses the course had was the loss of in-person TA verification. This could be alleviated with a Verify Lab switch that adds a "Test Output View" to the editor, which upon simulation, will contain the output from a separate simulation with a testbench intended for that lab, which would report what exactly was wrong with the code.
- This also reduces the need for TAs to have to do menial things like code grading.
- This may prove problematic to some students whose code simply just does not simulate correctly on CVC. As mentioned above, CVC could either be updated or replaced during development over the summer.

Remote view option (alias LiveDebug!)

- This would implement a live code-view system in which a student can “share” their code with a TA directly through the simulator. Using a separate video call app to communicate if needed, they can collaboratively type into the same code editor, with the other’s cursor moving in real-time as it makes edits to the code.
- Such a feature would be incredibly useful in this new era of remote-viewing and support.

Local client versions of the simulator for users

- A highly popular idea was the idea that the simulator should be a downloadable product. This was initially dismissed because the software used only ran on Linux, and the simulator includes far too much integrated software that would be extremely unnecessary to running it on a user’s computer (Redis, logging practices, admin panel, etc.)

- However, when the code gets cleaned up, and if the software changes so that it is available on all platforms (Windows, macOS, and Linux) this idea could be implemented.

Compiler output viewer

- Since it is crucial that students understand what is happening with the Verilog synthesis tool, one could add a permanent tab called Tool Output View, that would show the output of Yosys after synthesis, regardless of errors in the process. This would give students and grading TAs necessary context from which they can understand the reasons behind why their errors occurred.
- For very long errors that are annoying to type out, it could be problematic to try to hover and take a screenshot at the same time. To alleviate this, errors should be made sticky so that clicking on it will keep it in view to allow the screenshot to take place more easily.

Waveform viewer

- The primary issue here would be creating a HTML5-based VCD file renderer. There is no quick fix or module that could be installed for this situation, so one would have to write a parser, and subsequent waveform generation tool in JavaScript to implement such a feature. To avoid crowding the page too much, this could be placed on an entirely new page (e.g. `/waveform`).

Tutorial system

- Although probably not crucial, it could definitely be helpful for new users - when they log in for the first time, they will be introduced to a full walkthrough of how to simulate some Verilog code, and if they'd like to learn it, a step-by-step walkthrough of the invaluable keyboard shortcuts to speed up coding.

Acknowledgements

- The author’s students and fellow TAs in ECE 270, who provided insightful feedback on behavioral issues with the front-end.
- The developers maintaining the Yosys synthesis tool over at GitHub who are working extremely hard to introduce more useful features every day.
- Rick, whose continued guidance, presence, and enthusiasm helped me make the simulator better with each revision.