RHL-Butterfly: A Scalable IoT-Based Breadboard Platform for Embedded Systems and Remote Laboratories

Matthew Guo

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2023

Committee:

Rania Hussein

Shwetak Patel

University of Washington

Abstract

RHL-Butterfly: A Scalable IoT-Based Breadboard Platform for Embedded Systems and Remote Laboratories

Matthew Guo

Chair of the Supervisory Committee:

Rania Hussein

Department of Electrical & Computer Engineering

The RHL-Butterfly is a research to practice virtualized breadboard solution for FPGAs and ARM microcontrollers in remote laboratories and engineering education curriculum. The COVID-19 pandemic brought challenges to traditional engineering education practices, particularly with hardware, hands-on engineering practices without compromising creativity and instruction. The RHL-Butterfly aims to address traditional engineering education shortcomings and provide equitable access for all students interested in the engineering curriculum. This work improves upon an existing virtual breadboard model by using virtualization to interface a virtual breadboard with physical, remote hardware from a website user interface and presents a solution to support FPGAs and ARM microcontrollers and supporting intermediate logic gate integrated circuits. The new iteration of the virtualized breadboard uses a custom protocol that converts the graphically represented breadboard layout into a 1D string representation for network communication. The 1D string representation is then parsed in a custom designed, open-source, and scalable breadboard parser for embedded systems. This balance between a virtualized interface and physical hardware implementation preserves a hardware curriculum embedded systems engineering education and brings a promising solution to expand the scalability and accessibility of engineering labs.

Acknowledgements

I would like to thank everyone that I have worked with throughout my Master's degree program.

First and foremost, I would like to express my deepest gratitude to my advisor, Professor Rania Hussein, for her unwavering support and mentorship during my graduate school journey. From the inception of the Remote Hub Lab, she gave me the opportunity to be a part of it, which has been instrumental in shaping this thesis work. Working alongside her in the lab and as a teaching assistant for the digital design course has not only inspired my research, but also enabled me to gather survey data for the initial development of the RHL-Butterfly, leading to a second iteration of the virtual breadboard with improved refinement during another course offering. Beyond technical skills, she has provided me with invaluable guidance throughout my Master's, and I am truly grateful for her continuous support. I would also like to extend my heartfelt appreciation to Professor Shwetak Patel for serving on my thesis committee and providing invaluable feedback on RHL-Butterfly, highlighting its potential to make a significant impact in engineering education.

I am incredibly grateful to Dr. Pablo Orduña, co-founder of LabsLand, our industry partner for the RHL-Butterfly project. Dr. Orduña's mentorship and provision of necessary resources and tools for the development and deployment of the project, have played a vital role in expanding the usage and impact of this project to educational institutions globally through the LabsLand network. The strategic partnership with LabsLand has been instrumental in bridging the gap between theoretical research and practical implementation, allowing us to turn the butterfly project from a mere research endeavor into a practical reality.

Special thanks to Intel Corporation for funding this project.

Many thanks to each and every member at the Remote Hub Lab. My two years at the Master's degree program at the University of Washington is highlighted by the relationships and friendships that I have made for all of the lab members. Special thanks to Pedro Amarante and Jared Yoder, who have assisted with the testing phase of the virtual breadboard prior to deployment, Marcos Inonan, Brian Chap, and Francisco Luquin Monroy, whom I have become great friends with in the lab, and Stefhany Alves and Sai Jayanth Kalisi, who have helped run the logistics within the Remote Hub Lab.

Thanks to ECE faculty, staff, and students for the support for my educational journey at the University of Washington. Also, thanks to Professor Justin Hsia of CSE for allowing me to be his teaching assistant twice for the digital design course, one of them was during a critical time of deploying RHL-Butterfly, which needed the collection of more data from students for additional virtual breadboard refinements. I extend my sincere appreciation to all the students I had the privilege of being a teaching assistant for. Their enthusiasm, dedication, hard work, and willingness to voice their opinions toward the virtual breadboard and toward the course

curriculum greatly contributed to the success of the digital design course and the development of the RHL-Butterfly.

Finally, I want to thank all my friends and family members for providing me with the strength and the motivation to keep going, and persevering through even the hardest times during my program. A special shoutout and heartfelt gratitude to my mom, whose unconditional love and unwavering support have played an immense role in shaping me into the person I am today. My achievements are not and should not be considered solely my own, but shared through all those that have given me support throughout my life.

TABLE OF CONTENTS

Chapter 1. Introduction 2

1.1 Remote Laboratories 2

1.2 Existing Virtual Breadboards 2

1.3 Previous Iteration 4

Chapter 2. Methodology 6

2.1 Frontend 6

2.2 Backend 27

Chapter 3. Results 36

3.1 Update Graphs 36

3.2 LabsLand Integration 44

3.3 Integration to Curriculum 47

Chapter 4. Future Work 61

Chapter 5. Conclusion 62

Chapter 6. References 63

Chapter 1. INTRODUCTION

1.1 REMOTE LABORATORIES

In early 2020, when institutions across the nation were forced to cancel in-person learning in favor of a remote one, online instruction practices for how students handle physical prototyping hardware also needed to pivot without significantly hindering engineering curriculum and student learning. The idea of virtualizing remote laboratory equipment is not new but has become a much larger topic of experimentation and implementation due to the recent circumstances of the global pandemic [1][2]. Recent research on virtualizing hardware practices have favored simulated and remote laboratory instruction, due to benefits in financial savings, increased safety for potential dangerous electrical experimentation, and improving hardware accessibility to handicapped learners [3], with little to no significant difference in educational outcomes for the students [4][5]. Further research at the University of Washington has indicated that curricula that involve remote laboratory hardware yield higher student analytical learning experiences over the same curriculum that uses traditional physical laboratory hardware [6]. A different study for the same curriculum a few academic terms later analyzed student perspectives on using remote laboratories for their homework assignments, and found positive results for remote laboratories' convenience, ease of use, and financial savings [7].

Demand for remote engineering platforms is only expected to increase with the growth of stay-at-home policies in both industry and education. Just as many industries in a post-pandemic world are implementing and embracing a fully remote or a hybrid model working environment, the educational system should similarly present tools and options to allow remote and in-person learning opportunities that widens student accessibility to learning; greater opportunities for asynchronous learning are needed for students that face unexpected factors that prevents in-person education [8] throughout an academic term. In one study, 80% of students who have previously experienced some form of remote learning expressed interest in seeing a continuation of some online instruction in a post pandemic world [9]. The Remote Hub Lab [10] is one that is dedicated to this purpose, with a wide variety of remote education technologies that provide remote learning tools for when the education curriculum needs it.

1.2 EXISTING VIRTUAL BREADBOARDS

The rapid transition and quick curriculum changes to tailor toward online and remote variants of the course brought about many challenges. Universities and education institutes across the globe were poorly prepared for an emergency transition toward remote learning, with limited learning tools developed and fewer understandings of the available teaching and remote learning resources around them [11]. Engineering curriculum, particularly those that required students to

develop and create designs using laboratory equipment, quickly had to pivot their teaching methods while continuing the "learning by doing" philosophies from traditional in person education. One method that was widely adopted was providing students with take-home kits that consisted of key laboratory equipment throughout the quarter, which included taking home breadboards and core components. To demonstrate a proper breadboard circuit build, students used their smartphone cameras to take a video and demonstrate that the hardware they had built was correct. A survey taken at the end of the semester had analyzed student satisfaction with the emergency preparedness of the remote course offering, to which students largely complained about the inadequate platform to seek help with breadboard debugging, as many students resorted with taking pictures and sharing Zoom live feeds of their breadboard prototyping experience [5]. While using take home laboratory kits continue to promote a "learning by doing" approach, the difficulty in the debugging process from this approach cannot be undermined, with a large part of the frustrations of the debugging process stemming from a lack of adequate remote laboratory tools that properly interface with real world hardware.

Truly simulated virtual breadboards are available on the market today. One famous and ubiquitous virtual breadboard simulator is the Java Digital Breadboard Simulator, developed by Nicholas Glass for a thesis project in the University of York Computer Science in 2002. Featuring an extensive component library catalog, this virtual breadboard simulator is designed for circuit level simulation, logic level simulation, and functional level of simulation. The project understandably acknowledges that a truly virtual breadboard experience must make simplifications in the simulation process to save computation [12]. The virtual breadboard, while possessing extensive libraries, does not offer a way to interface an external breadboard design with microcontrollers or microprocessors. Tinkercad, another popular existing virtual breadboard simulator, is one such simulator capable of interfacing with microcontrollers [13], and was used by Professor Paola La Rocca at the University of Catania in response to the global pandemic to teach Arduino programming and hardware interface virtually. The virtual simulator uses a virtual breadboard that interfaces with a virtual Arduino to learn the basics of Arduino sketches [14]. Like the Java Digital Breadboard Simulator, Tinkercad is purely simulation-based that requires no physical hardware.

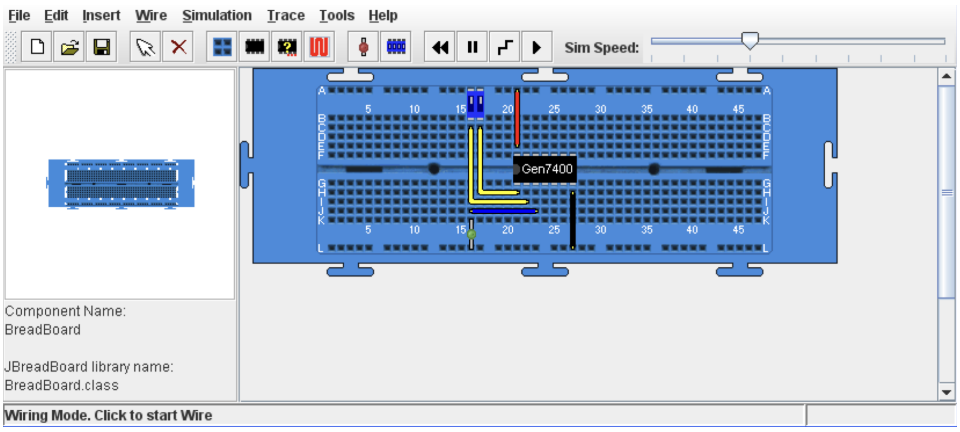

Figure 1.1: Java Digital Breadboard Simulator user interface. [12]

These existing virtual breadboards that are on the market today, or the take-home laboratory kit strategy used by different universities, all offer different perspectives as to how breadboard prototyping can be used in a virtual setting. Depending on the engineering goals in a particular university or engineering course, these practices may be adequate for student learning on the material. However, none of these breadboard implementations offer a way to interface a virtual and simulated breadboard experience to real world hardware, which would allow the preservation of real-world hardware responses while simultaneously offering easier ways for students to access, share, and debug breadboard designs for their course curriculum. LabsLand, a company that was formed from a spin-off of the WebLab-Duesto research group at the University of Deusto, acknowledged the lack of technical framework of, business initiatives in, and open-source contributions to a fully accessible and shared remote laboratory hardware infrastructure between multiple universities around the globe [15]. Through a partnership with LabsLand and the utilization of the LabsLand vast remote laboratory hardware infrastructure, a new virtual breadboard can be created that interfaces with real world hardware, which allows a virtual breadboard to be less of a digital simulation tool and more of a way to virtualize real hardware, and provide additional educational tools for engineering courses looking into using remote laboratories.

1.3 Previous Iteration

In Autumn 2020, a junior-level level course focusing on Hardware Description Language (HDL) and using Field Programmable Gate Arrays (FPGA) was converted to an online curriculum amid the global pandemic. This conversion presented a pilot project on the feasibility of using remote laboratories, and the viability of continuing remote laboratory education for a post pandemic world [6]. The first FPGA design lab assignment in the course aims to be a refresher of finite state machines (FSM), a concept learned in prior, prerequisite courses, and the introduction to the onboard general-purpose input/output (GPIO), which used a first iteration virtual breadboard

interface to simulate circuit buildings and interfacing GPIOs to students' SystemVerilog implementations on the FSM [16]. Traditionally with physical hardware, students receive a breadboard, an Intel DE1-SoC FPGA, physical switches, current limiting resistors, and light-emitting diodes (LEDs) to interface with the GPIO of the FPGA for their FSM design, as shown in the figure below.

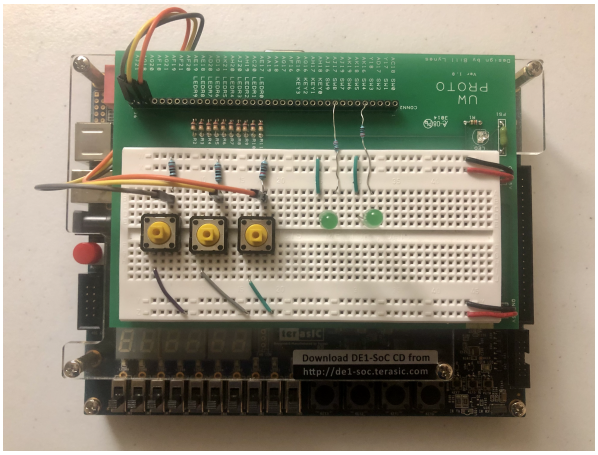

Figure 1.2: Physical breadboard construction of students' Lab 1. [15]

When pivoting to remote learning amid the pandemic, the physical breadboard was instead replaced with a virtual one. This virtual breadboard uses the Virtual Instrument Systems in Reality (VISIR) JavaScript framework for the first iteration virtual breadboard interface [17] to interface with a global network of real FPGA DE1-SoC hardware through LabsLand [15]. In the initial course offering of the remote breadboard interface, students were asked to participate in an anonymous online survey indicating the usability and practicality of the virtualized breadboard based on a 5-point Likert scale, which yielded favorable results for the breadboard user interface (UI) design and ease of use, but noted that the capabilities and interface had room for growth and improvements [16]. The previous iteration provides statically placed Single Pole Double Throw (SPDT) switches and virtual breadboard LEDs, which provides simplicity for students for the laboratory assignments, but would not easily scale with additional course curriculum or experimental design projects. The promising results of the previous iteration provided motivation for the expansion of breadboard digital logic features in RHL-Butterfly, where "RHL" stands for the Remote Hub Lab [10], such as the addition of digital logic integrated circuits and freedom for component placement to remote laboratories, which is highlighted in this thesis.

Chapter 2. METHODOLOGY

The RHL-Butterfly brings communication between a virtualized breadboard with a physical target hardware. The virtual breadboard solution needs to be both electrically accurate in a physical breadboard and have fast communication to the microcontroller hardware to justify the product design and provide a viable remote alternative to physical laboratory breadboards. These metrics define the user-experience for remote laboratories; noticeable differences with real-world hardware can provide distractions to the user experimenting their creations, and may inhibit the effectiveness of remote laboratories and their abilities to learn.

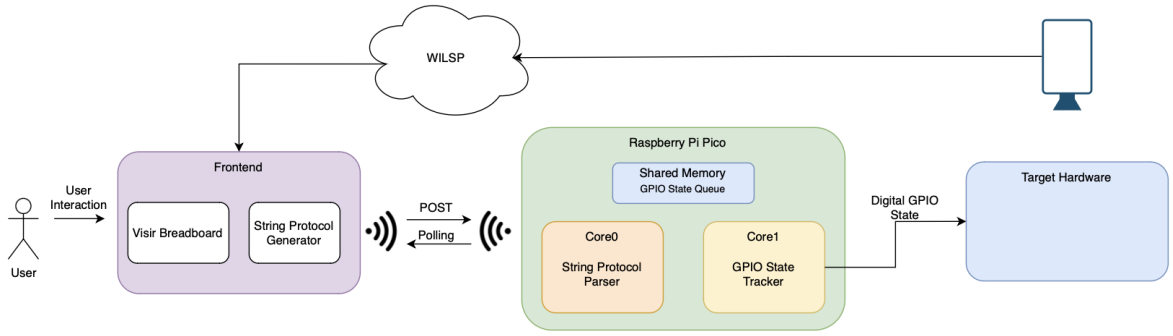

Figure 2.1: Top level block diagram of RHL-Butterfly.

At its core, the virtual breadboard experience and the system level communication is broken down into three distinct phases: a frontend phase, written in JavaScript, which provides the user with a virtual breadboard user interface to interact and add breadboard components, a Raspberry Pi Pico backend phase, which computes the digital logic states from the user submitted virtual breadboard design, and the target hardware phase, which is the microcontroller or FPGA that the users interface the virtual breadboard with for their coursework. The response from the target hardware is captured via a livestream that is then transmitted through WILSP [18] back over to the frontend to complete the remote laboratory experience with minimal latency.

2.1 FRONTEND

The breadboard builds upon the VISIR JavaScript framework that allows wiring and measuring electronic circuits through a virtual platform [19][17]. Users of this platform are presented with a virtual image of a solderless breadboard with the ability to draw wires freely on the breadboard through a clickable mouse drag.

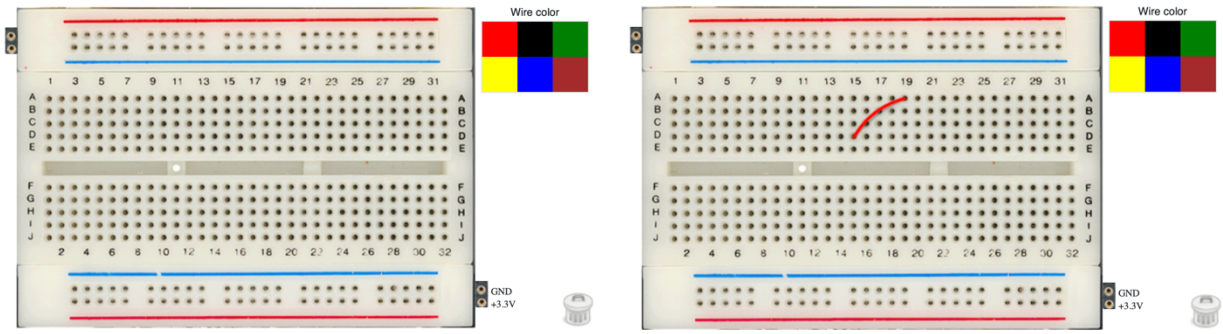

Figure 2.2: a) VISIR empty breadboard (left) and b) breadboard with drawn wire (right).

The VISIR virtual breadboard framework supports six different choices in wire color to aid in wire creativity and allow users to set breadboard experimentation organization that uniquely works for them. When a wire color is selected, the mouse then acts as a pen to draw a wire from a starting location to and ending one, provided that the user continues to press down on their mouse click. Once the mouse click is released, the wire drawing is finalized and locked in place to the virtual breadboard.

2.1.1 COMPONENTS

The goal for the improvement of the existing virtual breadboard is the inclusion of many different common digital logic components, allowing the breadboard to be used in many introductory to intermediate digital logic design curriculum. Of the seven basic logic gate components (AND, OR, XOR, NOT, NAND, NOR, and XNOR), four were identified as the fundamental building blocks for digital logic gates (AND, OR, NOT, and XOR), and thus were supported for the newest iteration of the virtual breadboard. Virtual actuators and transducers, in the form of dual-state switches and virtual LEDs, are also supported.

2.1.1.1 Logic Gates – Single Input

A NOT gate is the only basic logic gate component with a single input, with the basic functionality of inverting incoming digital signals. A NOT gate follows the following truth table:

Input	**Output**
HIGH	LOW
LOW	HIGH

Table 2.1: NOT gate logic table.

Typical Dual Inline Package (DIP) NOT gate Integrated Circuits (IC) purchased from online vendors, electrical component stores, and retailers consist of not one single NOT gate, but multiple. A common example for a NOT gate IC is the 7404 NOT Gate Package, consisting of 14 pins and six unique NOT gates. The input supplied voltage pin, or Vcc, is typically labeled as pin 14, while the associated electrical ground (GND) pin is typically pin number seven. Pins one, three, five, nine, 11, and 13 are associated inputs to pins two, four, six, eight, ten, and 12, respectively.

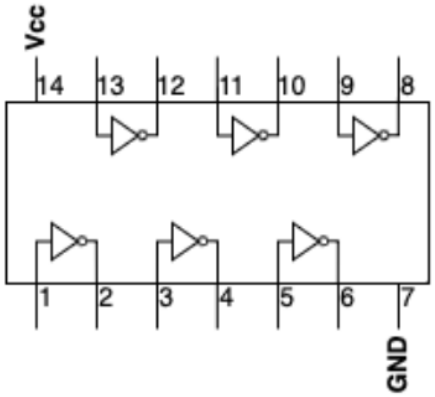

Figure 2.3: NOT gate 14-pin 7404 topology.

The NOT gate 7404 virtual breadboard equivalent component was created using an image of the unique 14-pin integrated circuit package, with a “NOT” label on top for the users to identify.

Figure 2.4: NOT gate image for the virtual breadboard.

The NOT gate functionality is handled through a JavaScript object, nested within a Breadboard object, which is the object container for the overall virtual breadboard. To ensure the functionality that users can click and drag the NOT gate component anywhere on the virtual breadboard, mimicking the freedom of integrated circuit placement found in a physical breadboard counterpart, the NOT gate JavaScript object stores the current left position (`this._leftPosition`) and top position (`this._topPosition`) pixel values of each created NOT gate. Whenever the user drags the component to a different location on the virtual

breadboard, a `setPinLocation` function is automatically called to update the stored pixel values.

2.1.1.2 Logic Gates – Dual Input

Aside from a NOT gate, the other three fundamental logic gate building blocks are dual inputs, thus taking two inputs to calculate the associating output.

For an AND gate, the output will only a digital logic HIGH if and only if both input1 and input2 are HIGH. If one or more of the inputs are not HIGH, the associated output remains a digital logic LOW. As such, the AND gate follows the following truth table.

Input 1	**Input 2**	**Output**
LOW	LOW	LOW
LOW	HIGH	LOW
HIGH	LOW	LOW
HIGH	HIGH	HIGH

Table 2.2: AND gate truth table.

For an OR gate, the output will only a digital logic LOW if and only if both input 1 and input 2 are LOW. If one or more of the inputs are not LOW, the associated output remains a digital logic HIGH. As such, the OR gate follows the following truth table.

Input 1	**Input 2**	**Output**
LOW	LOW	LOW
LOW	HIGH	HIGH
HIGH	LOW	HIGH
HIGH	HIGH	HIGH

Table 2.3: OR gate truth table.

An exclusive OR, or XOR, logic gate will output a digital logic HIGH if and only if input 1 and input 2 are unequal. If both input 1 and input 2 are equal, the associated output remains a digital logic LOW. As such, the XOR gate follows the following truth table.

Input 1	Input 2	Output
LOW	LOW	LOW
LOW	HIGH	HIGH
HIGH	LOW	HIGH
HIGH	HIGH	LOW

Table 2.4: XOR gate truth table.

Similar to that of a NOT gate, typical DIP logic gate integrated circuits consist of more than a single gate. If a 14-pin DIP package is used, one pin is used for Vcc (typically pin 14), another is used for GND (typically pin 7), and the rest are used for the logic gates, each with three pins (two input pins and one output pin), thus totaling to four distinct gates.

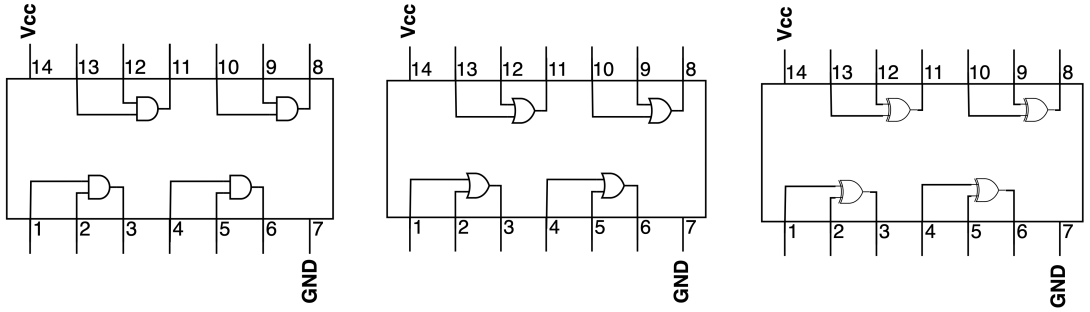

Figure 2.5: a) AND gate 14-pin 7408 topology (left), b) OR gate 14-pin 7432 topology (middle), and c) XOR gate 14-pin 7486 topology (right).

Despite differences in logic gate functionality, a 7408 Quad 2-Input AND gate, a 7432 Quad 2-Input OR gate, and a 7486 Quad 2-Input XOR gate the same associated pin assignments: input 1 is associated with pins one, four, ten, and 13, input 2 is associated with two, five, nine, and 12, and the outputs are associated with pins three, six, eight, and 11, respectively.

Figure 2.6: a) AND gate image for the virtual breadboard (left), b) OR gate image for the virtual breadboard (middle), and c) XOR gate image for the virtual breadboard (right).

Because they share similar pin assignments, the functionalities of the JavaScript object that store each of the different logic gates are similar, and as such inherit from a larger and more general `QuadDualInputGate` JavaScript object. Similar to that of the NOT gate, in order the ensure the ability to freely drag and place these logic gate components anywhere on the virtual breadboard, the larger `QuadDualInputGate` component stores the current left position and top position pixel values in an internal variable, which is updated every time the component is dragged to a different location through a prototype function `setPinLocation`.

2.1.1.3 Virtual Switches

A Single Pole, Double Throw (SPDT) dual state toggle switch changes its electrical state through the mechanical movement that opens or closes an electrical circuit. Typically, one side of the switch is connected to a power plane, such as Vcc, while the other side is connected the electrical reference GND. The middle pin is then connected to the rest of a user's designed electrical circuit. When a user mechanically toggles the state of the switch, the middle pin then is toggled to either connect to Vcc, indicating a digital logic HIGH, or to GND, indicating a digital logic LOW.

For a virtual breadboard, users are unable to physically interact with the mechanical movement of a SPDT switch. Instead, to create the illusion of a virtual switch changing its state, a JavaScript switch component stores two different images: one indicating an ON position, and another indicating an OFF position.

Figure 2.7: a) An SPDT virtual switch to the left position (left) and b) an SPDT virtual switch to the right position (right).

Each of these switches contain three pins: a left pin, a middle pin, and a right pin. The left pin is typically used to connect with one digital logic state (i.e. HIGH or LOW), while the right pin is typically used to connect with the other digital logic state. The middle pin is then used to connect

with the rest of the circuit. When a user clicks on the virtual switch, the JavaScript component immediately changes the image to the other position, thus mimicking the behavior of mechanically moving the switch state.

As with the other previously mentioned components, to ensure that users can drag and drop each virtual switch to their desired location on the virtual breadboard, the virtual switch JavaScript object stores the left and top pixel value positions internally in that object. These values are updated whenever the switch is dragged to a new location on the virtual breadboard. However, unlike the logic gate components, the output pin digital logic state is computed in the client frontend, rather than through a microcontroller on the backend. Because the JavaScript must flip the image of the virtual switch during a mouse-click event trigger, the digital state of the virtual switch is computed regardless.

2.1.1.4 Virtual LEDs

Similar to that of a virtual switch, a virtual LED stores two different images to create an illusion of the transducer changing its state: one illuminated image, and another unilluminated image. These images and the virtual LED functionality are stored in a JavaScript object, similar to the previously mentioned components. However, unlike a virtual switch and virtual logic gate integrated circuits, LEDs can change its orientation while still being electrically correct on a solderless breadboard, albeit dependent on the location to which the virtual LEDs are placed. To provide a similar amount of customizability to that of a physical breadboard design, the virtual LED can be rotated 90 degrees by the user.

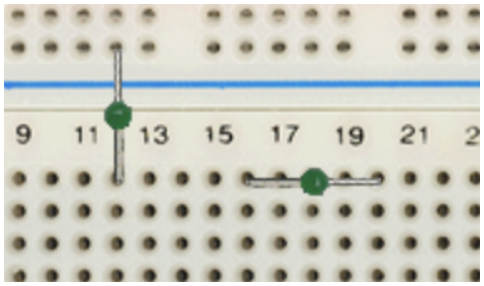

Figure 2.8: Virtual LED placed virtually and horizontally.

The virtual LED JavaScript object stores the left and top positions of the image pixel values to incorporate the draggability feature found in the other supported virtual breadboard components. When the user drags the virtual LED to a new location, the left and top position values are updated accordingly to ensure that the JavaScript object always stores the latest object placement information. However, in order to support the virtual LED functionality for both vertical and horizontal orientations, the JavaScript object stores an additional Boolean value within the

object: `this._isVertical`. This Boolean value asserts true when the virtual LED is in the vertical position, and is deasserted when the virtual LED is in the horizontal position. The top and left position values of the LED are updated differently depending on that Boolean value.

2.1.2 ADDING COMPONENTS

The list of supported components for users are intentionally kept out of view so that the user can focus on the breadboard design itself. When the user desires to add additional components to their design and is ready to view the list of available components, this list of supported components are accessed through a tray, constructed by a hidden HTML div, displayed only when the user presses the "+" icon on the right of the virtual breadboard.

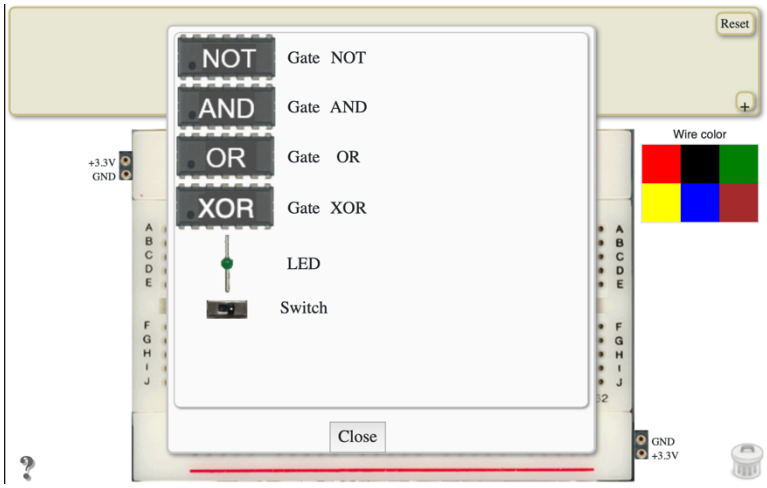

Figure 2.9: List of supported draggable components.

Users may click on a list of these supported components to add the component to the top left corner of the user interface, and then drag the components to the desired location of the breadboard.

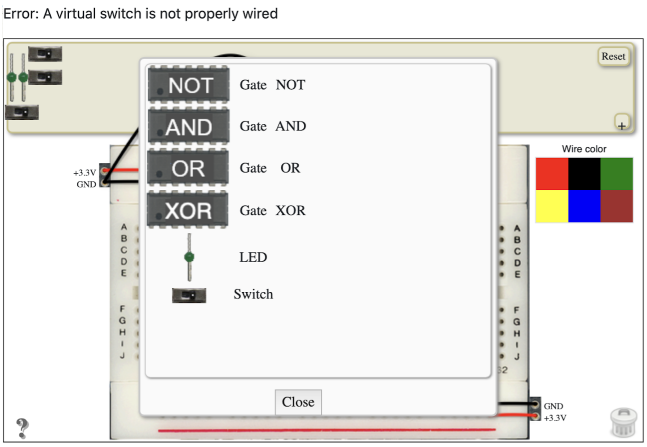

Figure 2.10: Adding components to the upper left corner of the breadboard.

For easy maintenance and scalability of newer supported components, the information on these components is stored in an XML file that the JavaScript automatically reads and parses once the "+" button is opened. The XML file contains a list of supported `<components>`, with each component dictating a component type, a name or value, the number of pins, and the file location for the breadboard component image. If the component supports rotation, such as the virtual LEDs, the component will contain multiple `<rotations>`, each initializing different x and y pin pixel values. Below is an example of a NOT gate component found in the XML file.

```
<components>
   <component type="Gate" value="NOT" pins="14">
       <rotations>
           <rotation ox="-18" oy ="-13" image="butterfly_not_gate.png" rot="0">
               <pins>
                   <pin x="-13" y="26" />
                   <pin x="0" y="26" />
                   <pin x="13" y="26" />
                   <pin x="26" y="26" />
                   <pin x="39" y="26" />
                   <pin x="52" y="26" />
                   <pin x="65" y="26" />
                   <pin x="65" y="-13" />
                   <pin x="52" y="-13" />
                   <pin x="39" y="-13" />
                   <pin x="26" y="-13" />
                   <pin x="13" y="-13" />
                   <pin x="0" y="-13" />
                   <pin x="-13" y="-13" />
               </pins>
           </rotation>
       </rotations>
   </component>
```

Figure 2.11: XML component for a NOT gate.

2.1.3 *Deleting Components*

Any user placed wires or components can also be deleted. Dragging a component is done by clicking and holding on a select component to place the component down to a desired location on the breadboard. For components that support a change in state, such as the virtual SPDT switch, this can be done by pressing on the component image.

To delete a component, a different user experience needed to be designed such that it does not interfere with the user experience of the other features, but also needs to be intuitive enough such that a component would not be deleted accidentally. The inspiration to delete a component comes from the default implementation of the VISIR framework to delete an already existing wire on the breadboard. Users would first click, but not drag, the desired wire to delete, which then highlights that wire. When the wire is highlighted, a trash icon to the bottom right corner of the user interface becomes active. Deleting the wire is done by then pressing on the delete icon.

This feature was incorporated to all the breadboard components. Users can click on any component already placed on the virtual breadboard to highlight that specific component and activate a trash icon. Once the user presses on the trash icon, placed specifically in the corner to eliminate accidental presses, the highlighted virtual breadboard component is removed from the breadboard UI.

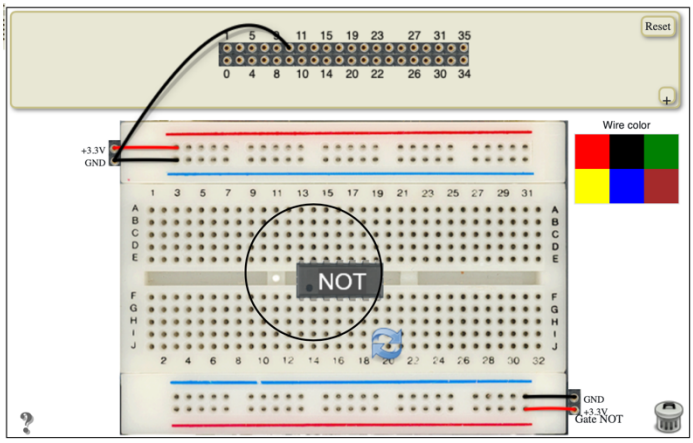

Figure 2.12: Selecting a NOT gate.

To ensure a quick way of clearing the entire virtual breadboard, a “Reset” button functionality is also added to quickly clear all components and wires on the breadboard.

2.1.4 *Protocol Construction*

The RHL-Butterfly offloads the virtual breadboard state computation to a backend microcontroller in order to better mimic real time behavior of an embedded system, thus saving

computation on the client frontend. To process the current breadboard state and the necessary information for the backend microcontroller, a custom string protocol of that breadboard design is constructed and passed to a Raspberry Pi Pico microcontroller through a POST request method to the microcontroller server.

This custom string protocol is a minimalistic way of describing the user designed circuit in as small of a string length as possible. This section describes the different elements of the virtual breadboard and its associating compressed string representation.

2.1.4.1 GPIO Header

The virtual breadboard can interface with a target microcontroller hardware using header pins that act as the GPIO of that target microcontroller. To simplify the protocol construction, certain GPIO pins are designated as only virtual inputs (target hardware outputs), while others are designated as only virtual outputs (target hardware inputs).

To provide information of whether a GPIO is connected, the protocol for a GPIO element on the virtual breadboard is a "g" + {gpio_number}. However, because the goal of the virtual breadboard is to interface with as many different microcontrollers and FPGAs as possible, each with different GPIO header sizes and pin mappings, the {gpio_number} is normalized to indicate not the exact header pin number, but rather in relation to the first GPIO input, or the first GPIO output. For example, if a target microcontroller GPIO contains 40 distinct pins, with the supported input pins starting at pin 23, and the next supported input pin at pin 24, rather than constructing the intuitive, but less scalable "g23" and "g24", the protocol is instead constructed to be "g00" and "g01".

Protocol Construction	**Example**
g{gpio_number}	g00

Table 2.5: GPIO Header protocol construction.

2.1.4.2 Power Plane

Vcc and GND are two supported power planes of the virtual breadboard. If part of a design is connected to a power plane, the protocol is constructed as "L" + {logic_level}, where the logic level is either True or False (T or F).

Protocol Construction	**Example**
L{logic_level}	Vcc: LT GND: LF

Table 2.6: Power Plane protocol construction.

2.1.4.3 Virtual Switches

Virtual switches are designed similarly as a power plane. Because the frontend JavaScript keeps track of the virtual switch ON and OFF position images to mimic a mechanical switch when the user clicks on the switch image, the state of the virtual switch is already computed, thus only the already computed state can be sent to the backend microcontroller.

The string protocol constructed is a "S" + {logic_level}, where the logic level is either True or False (T or F).

Protocol Construction	**Example**
S{logic_level}	HIGH: ST LOW: SF

Table 2.7: Virtual switch protocol construction.

2.1.4.4 Virtual LEDs

The virtual LEDs on the breadboard are able to be connected to outputs only, designed to be transducers of the electrical circuit. The backend microcontroller computes the digital logic state of all virtual LEDs on the frontend user interface and will send information to either display an illuminated LED image or an unilluminated LED. The client frontend constructs a string protocol to provide information of what the LED is connected to so that the microcontroller can then properly compute the LED state.

If an LED is present connected to a circuit on the virtual breadboard, the protocol constructed is "d" + {led_number}.

Protocol Construction	**Example**
d{led_number}	d0

Table 2.8: Virtual LED protocol construction.

2.1.4.5 Logic Gates

Depending on which logic gate is used, the logic gates can contain many different inputs and outputs. For a single input logic gate, such as a NOT gate, the string protocol constructed is "n" + {input} + {output}, where the input is a GPIO header protocol, a power plane protocol, or a virtual switch protocol, and the output is a GPIO header protocol or a virtual LED protocol. While a 7404 NOT gate integrated circuit contains many logic gates in the 14-pin package, the

information as to which exact logic gate number in the package is of little relevant information for the backend microcontroller, and as such, is abstracted out.

Component	Protocol Construction	Example
NOT gate	n{input}{output}	nSTg01

Table 2.9: Single input logic gate protocol construction.

For dual input logic gates (i.e. AND gates, OR gates, and XOR gates), the protocol constructed is a {gate_identifier} + {input1} + {input2} + {output}.

Component	Protocol Construction	Example
AND gate	a{input1}{input2}{output}	aSTLTg01
OR gate	o{input1}{input2}{output}	oSTLTd0
XOR gate	x{input1}{input2}{output}	xSFLFg02

Table 2.10: Dual input logic gate protocol construction.

2.1.4.6 No Logic Gates

A valid breadboard design does not need to have a logic gate. For example, a wire can be constructed that simply connects a virtual switch to a virtual LED. When this is the case, the protocol constructed is "y" + {input} + {output}. When the backend microcontroller receives a protocol containing a "y", it will immediately forward the logic state of the input to the output.

Protocol Construction	Example
y{input}{output}	ySTd0

Table 2.11: No logic gate protocol construction.

2.1.4.7 Buffers

Occasionally, a wire will be drawn from an output of a logic gate to an input of another gate. This is labeled as a buffer wire.

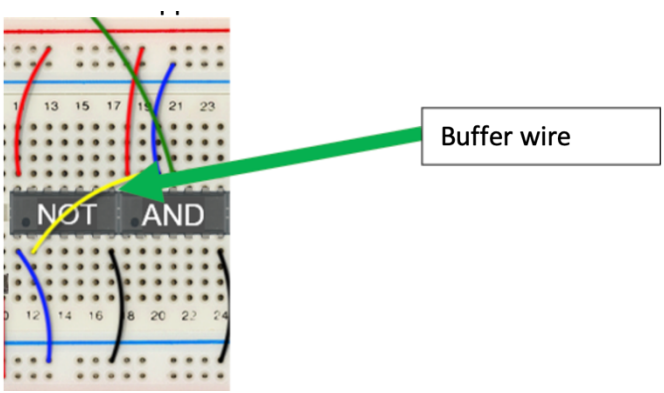

Figure 2.13: Virtual breadboard buffer wire.

When the protocol string is constructed, the frontend does not know the current digital state of the buffer wire. The backend microcontroller handles the computation of this electrical node when computing the digital state of the entire virtual breadboard. The frontend interprets that there is a need for a buffer wire and will construct the string protocol as “b” + {buffer_number}.

Protocol Construction	Example
b{buffer_number}	b0

Table 2.12: Buffer protocol construction.

2.1.4.8 Multiple Outputs

Valid designs for a breadboard also include multiple output wirings from a single output source. For example, the output of a NOT gate can be fed to multiple GPIO input pins. When this is a case, the client frontend separates each output with a comma “,”.

Protocol Construction	Example
{output1},{output2},…	nSTg00,g01

Table 2.13: Multiple output protocol construction.

2.1.4.9 Chaining Together Strings

To construct the string protocol that details the entire current breadboard state, the JavaScript sweeps through all user drawn wires.

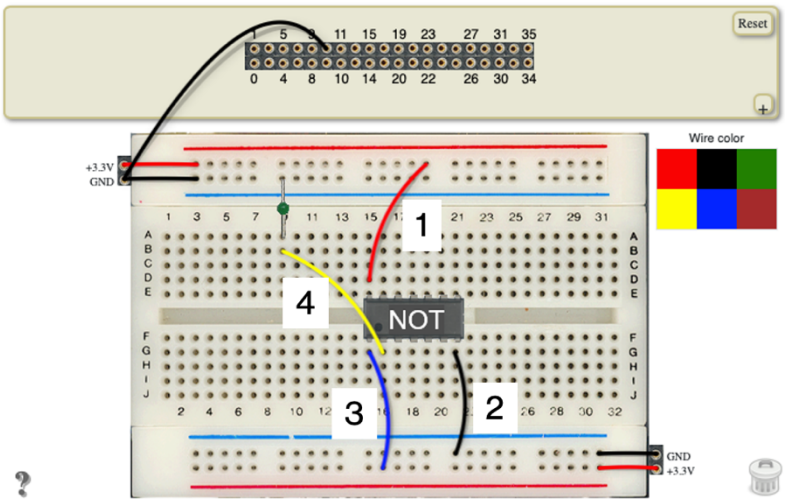

Figure 2.14: Sweeping through user drawn wires.

Each user drawn wire contains two endpoints. For the RHL-Butterfly, a valid design will only occur when a virtual breadboard defined input is connected to a virtual breadboard defined output.

Breadboard Inputs	**Breadboard Outputs**
Input GPIO	Output GPIO
Logic Gate Input	Logic Gate Output
Virtual LED	Vcc/GND
Buffer Wire	Buffer Wire
	Switch

Table 2.14: Virtual breadboard inputs and outputs.

For each wire, the frontend checks one wire endpoint to see if it is connected to an input. Then, it checks the same wire endpoint to see if it is connected to an output. Next, the frontend checks the other end of the wire point to see if that is connected to an input, and then checks that same wire end point to see if it is connected to an output. The client frontend will only proceed with the string protocol construction to be sent to the microcontroller if the first endpoint is connected to an endpoint, and the second endpoint is connected to an output, or vice versa. If a wire is drawn

that does not meet these requirements, the virtual breadboard interprets the design as an electrical error and prompts the user to accordingly.

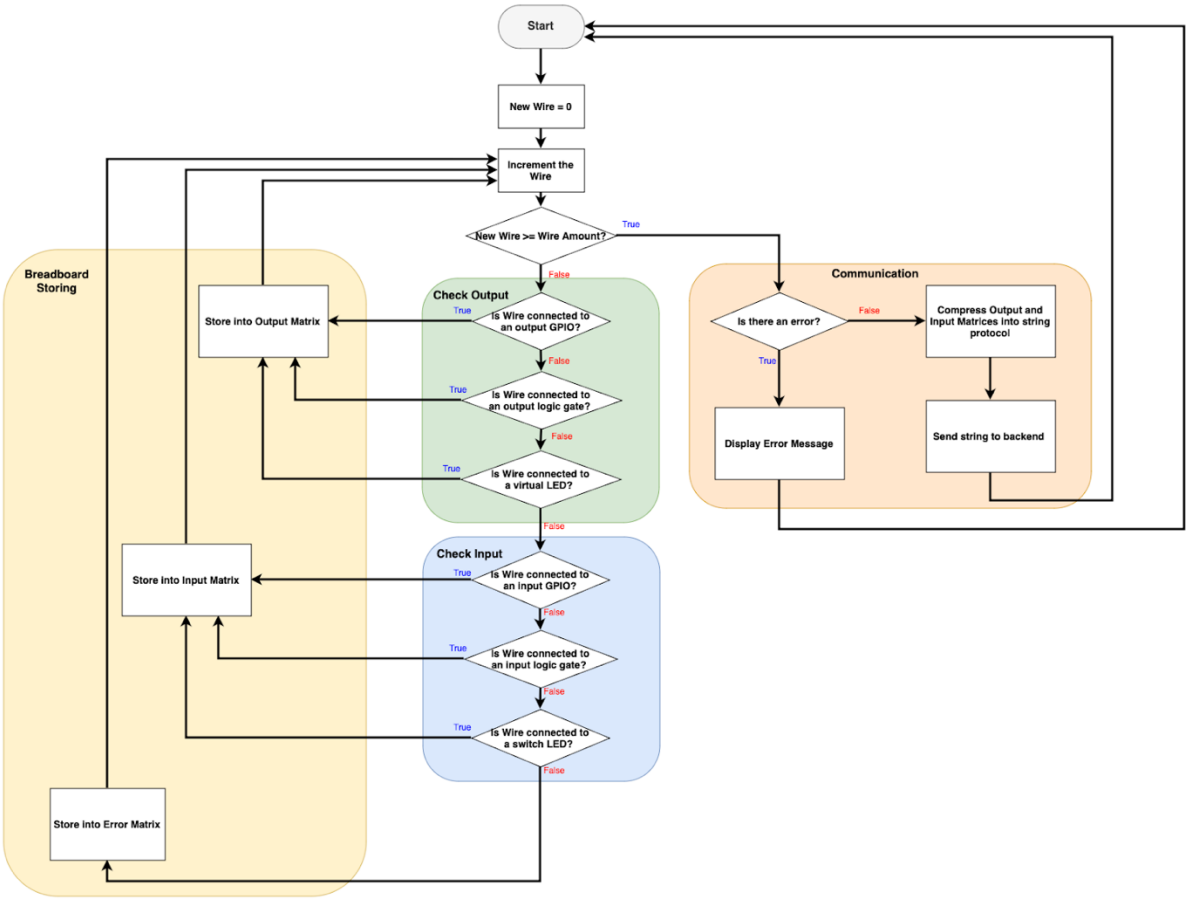

Figure 2.15: Flow diagram of JavaScript functionality.

Taking advantage of the mastery of the embedded system for asynchronous real-world detection, the breadboard GUI continuously updates its user-input state after every mouse click. This is intended to provide a similar experience as when prototyping with physical hardware; assuming that power is supplied to a physical breadboard design, electrical current will immediately flow through the elements when a component is placed in a valid location.

A safe and common workflow for breadboard prototyping would first be to create a circuit and verify the design before initializing power throughout the system for electrical testing. To preserve this workflow and force users to double-check their virtual breadboard design, the breadboard frontend would only submit the virtualized design when a user presses the “submit” button on the webpage, indicating that step of “powering” their design. When the “submit”

button is pressed and no design errors are found, the string protocol that is created then asynchronously is sent over to the Raspberry Pi Pico microcontroller for digital processing.

2.1.5 Error Messages

The backend microcontroller assumes a correctly built breadboard and a correctly generated string protocol from the virtual breadboard frontend. As the client frontend sweeps through the wires to generate the custom string protocol, the breadboard must first be electrically correct before the created string representation is sent over through a POST request to the microcontroller. As mentioned above, a valid circuit is one where all user placed wires are connected to a virtual breadboard defined input on one end, and a virtual breadboard defined output on the other. Any other configuration will assume an incorrect breadboard design, and thus generate an error message to the user. These error messages displayed by the client frontend are strategically worded such that they explain to the user what error has occurred, but does not inform the user how to solve the error. This is such that the experimentation and learning aspect of breadboard design can be preserved.

These assumptions for an input to only be connected to a valid output, while simplistic in nature, eliminates the possibility that a user submits an electrical short. For example, a Vcc power plane is defined as a virtual breadboard output. GND is also defined as a virtual breadboard output, and as such, a wire connection between Vcc and GND would trigger an invalid design. In a different case, an output of a digital logic gate is defined as a virtual output. When connected directly to GND, if the logic gate output is a digital logic HIGH, a short would occur and damage the integrated circuit due to exceeding the maximum rated source current. Although the integrated circuit would continue to function if the output was a digital logic LOW, without a current limiting resistor, the physical design would be an invalid one, which is represented in the virtual breadboard.

2.1.5.1 Output Connected to Output, or Input Connected to Input

If two outputs or two inputs are connected to each other, the frontend user interface stores the error and displays it on an HTML div to the user, asking them to check their wiring.

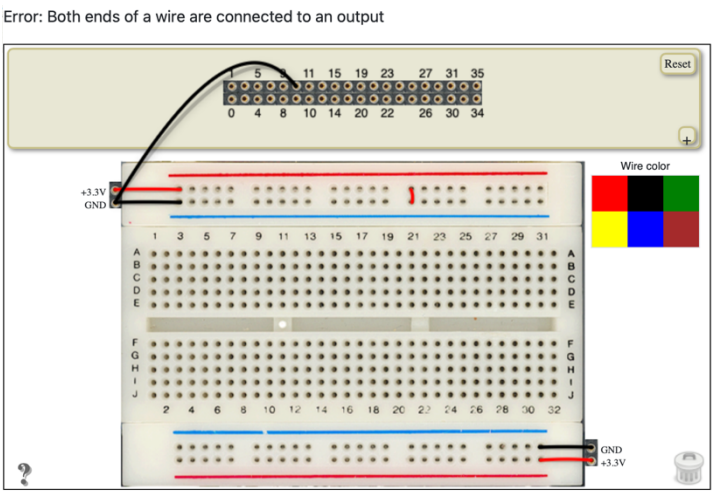

Figure 2.16: Virtual breadboard electrical short error.

2.1.5.2 Component Power

In a valid physical circuit, all logic gate integrated circuits must be properly connected to power (Vcc) and ground (GND) to ensure proper circuit functionality. To mimic that of a physical breadboard, the virtual breadboard also checks to ensure that every logic gate component is properly connected to power on its pin 14 and GND on its pin 7. The JavaScript tracks this by creating a `componentStatus` objects with two Boolean values: `connectedToPower` and `connectedToGround`. If a logic gate is used in the design, the `componentStatus` object is created and initializes both values as a Boolean False. When sweeping through the user constructed wires to generate the custom string protocol, these Boolean values are asserted if the JavaScript finds that a wire is properly connected to the components' pin 14 and pin 7. The string protocol for that logic gate will only be constructed if both `connectedToPower` and `connectedToGround` are asserted after sweeping through all wires in the breadboard design. Otherwise, the frontend returns a message to the user saying that a component is not properly powered.

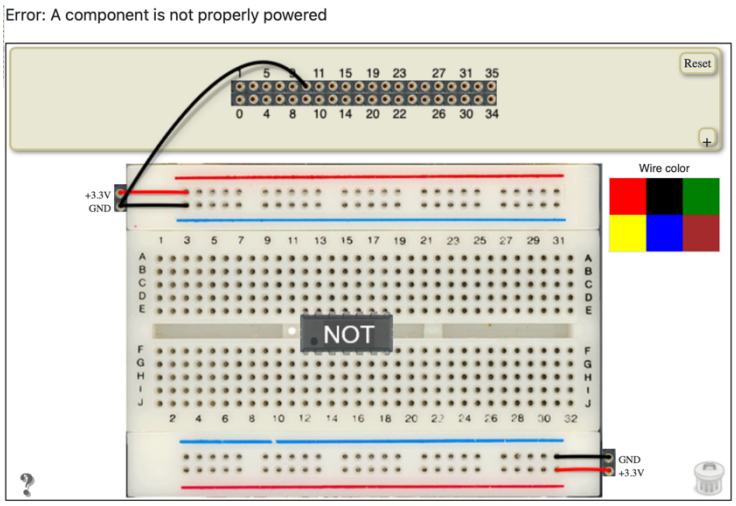

Figure 2.17: Virtual breadboard component power error.

2.1.5.3 Component Placement

The RHL-Butterfly allows users to drag any supported component to a user desired location on the virtual breadboard. This helps mimic the creativity of that of using a physical breadboard. However, due to the electrical characteristics of a solderless breadboard, integrated circuits have certain designated areas to be placed to ensure that the pins are electrically isolated from each other. An incorrect integrated circuit placement will create electrical failures and unintended electrical shorts in the breadboard design. The virtual breadboard checks the proper placement of the supported integrated circuits to ensure that the designed circuit is electrically correct. If the logic gate integrated circuit is not placed at the center horizontal nodes of the breadboard, the frontend user interface will display a message to the user that an error in the breadboard design has occurred.

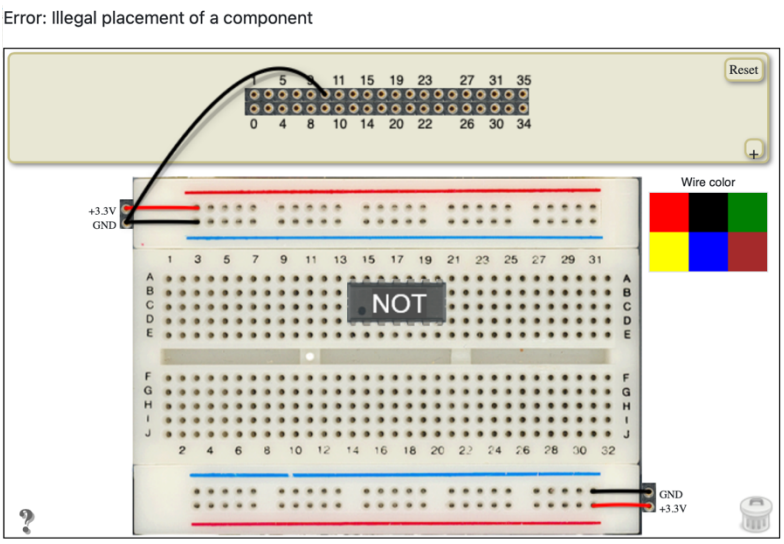

Figure 2.18: Virtual breadboard component placement error.

2.1.5.4 Virtual LEDs

Virtual LEDs are a supported component of the RHL-Butterfly. In the current iteration of this virtual breadboard, every LED must contain some distinct path to ground to ensure that Kirchoff's Current Law is enforced. If the LED does not contain an electrical current path to ground, the breadboard user interface produces an error, which is displayed for the user.

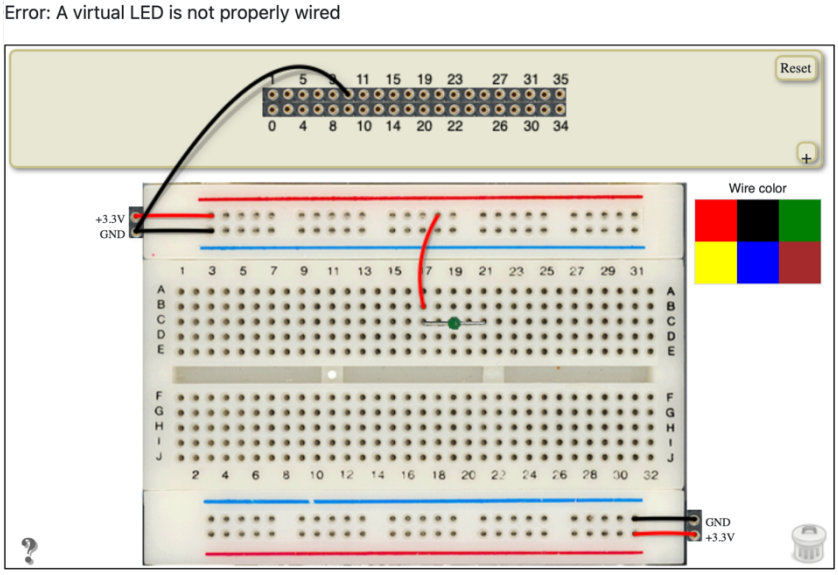

Figure 2.19: Virtual breadboard LED wiring error.

2.1.5.5 Virtual Switches

Dual state switches are supported components of the virtual breadboard. In the current iteration of this virtual breadboard, these switches are designed to be toggled between a logic HIGH and a logic LOW. To ensure that this is done electrically, one side of the dual-state switch must be connected to a power rail, while the other end must be connected to ground. If a switch is used in the design and is not met, the breadboard user interface produces an error message.

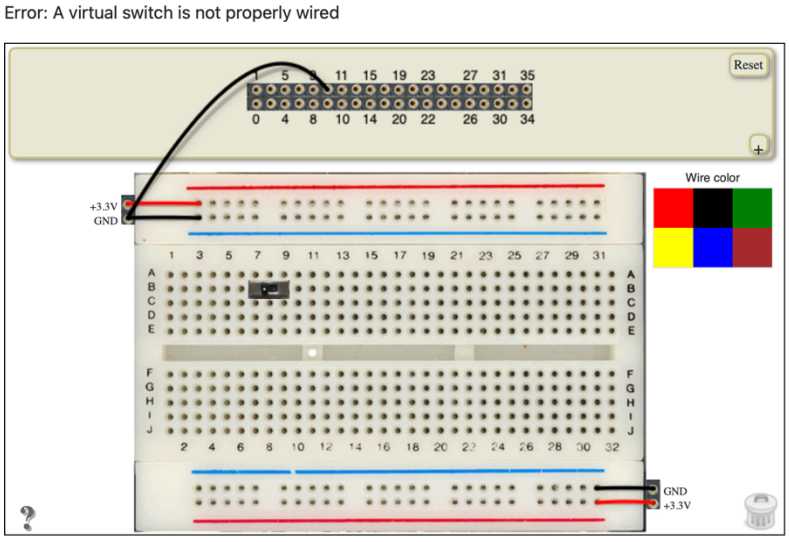

Figure 2.20: Virtual breadboard switch wiring error.

2.1.5.6 Null Components

One distinct error to the virtual breadboard that is not present in a physical one is the creation of additional electrical nodes. For simplicity in the protocol construction process, the frontend JavaScript sweeps through all user-drawn wires on the virtual breadboard and checks if a breadboard output is connected to a breadboard input. A byproduct of this is that all wires must be connected to some breadboard component. For example, one endpoint of a user drawn wire is connected to a logic gate, and the other endpoint is connected to a virtual LED.

In a physical breadboard, designers are allowed to construct additional electrical nodes for their designs, where the sole purpose of the newly created electrical node is to electrically jump to a component elsewhere on the breadboard. This creation of a new electrical node is not allowed on the virtual breadboard and will produce an error that is displayed to the user.

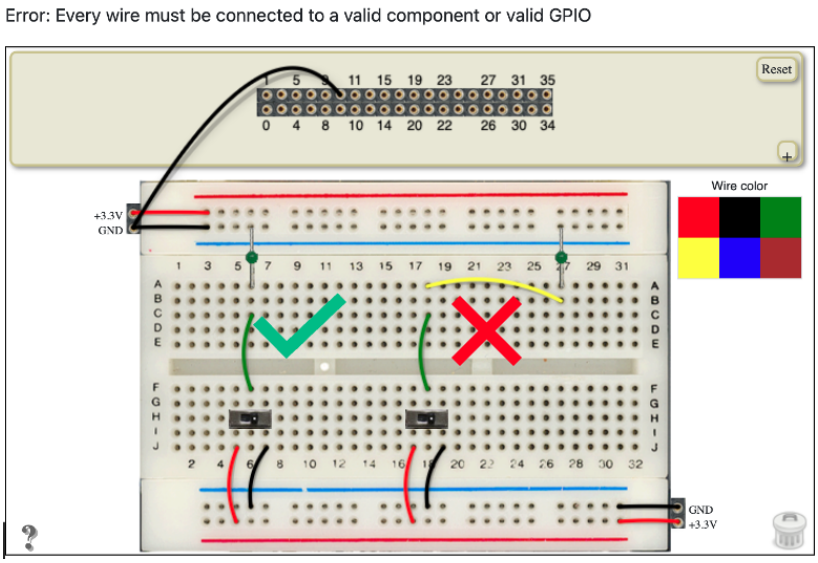

Figure 2.21: Virtual breadboard additional node creation error.

2.2 BACKEND

A Raspberry Pi Pico is used as the interpretation microcontroller that handles breadboard computation and translates the GPIO information directly to the target hardware. Having a consistent microcontroller hardware to conduct the serial protocol interpretation allows for scalability with other target hardware in the future. The Raspberry Pi Pico was chosen for the project due to its very high clock speed and for its affordable nature. It can clock up to 133MHz and supports multiple CPU cores for parallel processing, but defaults at a clock speed of 125MHz [20]. The Raspberry Pi Pico supports vast libraries through the PicoSDK [20] for experimentation and scalability, and future development on the microcontroller peripherals. It allows network communication through the web using an interfacing Raspberry Pi ecosystem.

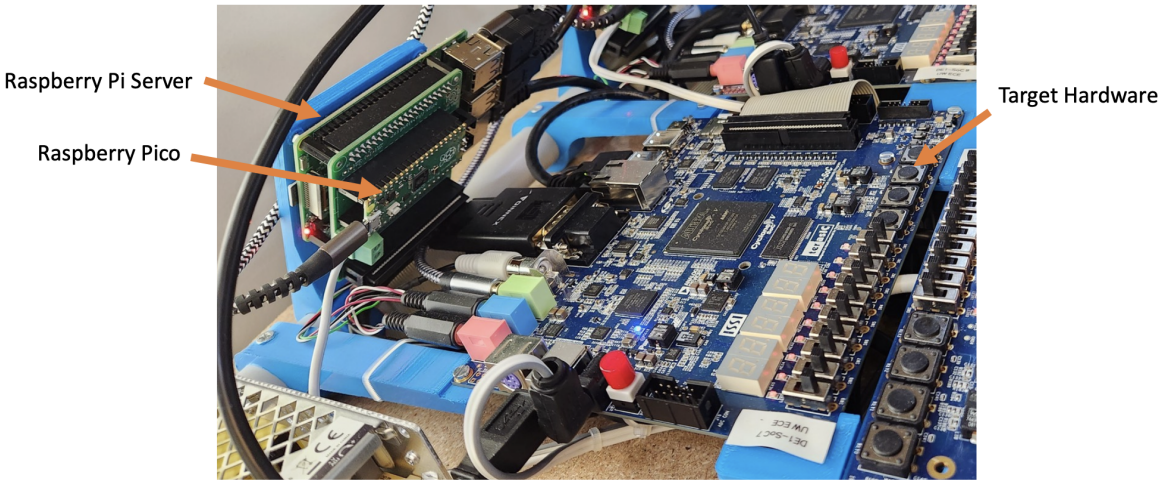

Figure 2.22: Communication hardware used for the remote laboratory hardware server.

2.3.1 Raspberry Pi Pico CPU Core Topology

The Raspberry Pi Pico features a dual core processor that enables the Pico module to be capable of multiple thread execution and multithreading. The Pico contains the processing cores Core1 and Core0, where the default Raspberry Pi Pico program is executed on Core0.

For the virtual breadboard backend, Core0 is focused on interpreting the string protocol that is sent by the JavaScript frontend. While parsing through the string protocol, the Raspberry Pi Pico computes the states of the GPIOs to be updated on the target hardware. For example, an output GPIO needs to be changed from a logic HIGH state to a logic LOW state. The change in digital logic level is stored onto a C++ vector array, which is then sent to a Raspberry Pi Pico queue.

Although Core0 and Core1 runs independent tasks, these cores share the same memory spaces. Thus, Core1 has access to the queue that Core0 updates, which contains the stored C++ vector array, detailing the change in the GPIO states. Core1 uses the C++ stored vector array to physically send the communication over to the target hardware to change the GPIO state. By running this separate from the main Core0, it lessens the load for one particular core; rather than have one core run both the string protocol interpretation and change the GPIO state of the target hardware, the task is then shared between the two available cores on the microcontroller.

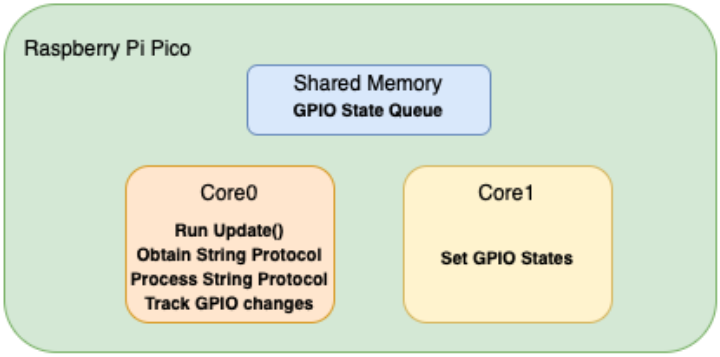

Figure 2.23: Raspberry Pi Pico Core0 and Core1 functionality.

2.3.2 PROTOCOL INTERPRETATION

The Raspberry Pi Pico uses shared memory to track the input GPIO states, output GPIO states, buffers, and virtual LED states. To enhance scalability and expandability for many different target hardware in the future, the JavaScript string protocol does not use specific GPIO pin numbers, but rather implements a counter based off the first available input or output GPIO. This same counter implementation is used to track buffers and virtual LED states of the breadboard. Due to this, it is advantageous to keep track of the states by using an array or vector. The counter number created from the string protocol construction in the JavaScript frontend acts as the index number for the vector or array, simplifying the ability to randomly access different array addresses if a change in the GPIO states is needed.

The JavaScript frontend client, when constructing the protocol, does not need to know the GPIO states after the Raspberry Pi Pico finishes its computation, as the information is not needed to process any changes in the frontend user interface. Rather, the GPIO state information is sent to the target hardware to directly change the physical GPIO state of that microcontroller or microprocessor. Similarly, neither the JavaScript frontend nor the target hardware needs to know the intermediate buffer states. These buffer states are used only to help aid in the computation of the GPIO states and virtual LED states. As such, the buffer states are stored in a vector in shared memory and is not passed anywhere else. What the JavaScript does need to know, however, is the state of the virtual LEDs, as depending on the virtual LED state, the image of the virtual LED may need to change in order to mimic the behavior of it being illuminated.

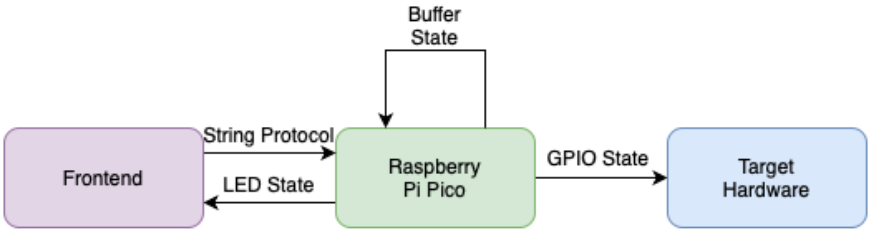

Figure 2.24: Raspberry Pi Pico information transfer.

An update function in Core0 is run to begin the JavaScript string protocol interpretation. If no changes in the frontend are made (i.e. the JavaScript string protocol is not changed), Core0 will process and interpret the last received JavaScript string protocol. Because the breadboard error checking is all done in the JavaScript frontend, the JavaScript will only send valid string protocols to the Raspberry Pi Pico. As such, the C++ firmware on the Raspberry Pi Pico assumes that any received JavaScript string protocol is a valid breadboard state.

The `update()` function sweeps through the characters in the JavaScript string protocol. Depending on the characters that it reads, the code will know how many inputs and the number of outputs there are for one string section. For example, if the update function reads a string protocol that describes a NOT gate, the code will then only need to handle one input. For a string protocol that describes an AND gate, the code will then need to handle two inputs.

String Protocol Character	Breadboard Associated Description	Number of Inputs	Number of Outputs
“n”	NOT gate	1	1
“a”	AND gate	2	1
“o”	OR gate	2	1
“x”	XOR gate	2	1
“y”	No gate	1	1

Table 2.15: String protocol interpretation for logic gates.

Within each input and output, the C++ code then parses the next character. Again, depending on the characters that it reads, the code will then know how many characters to read after that.

String Protocol Character	Breadboard Associated Description	Number of Inputs
“L”	Power or Ground Plane	1
“S”	Virtual Switch	1
“g”	GPIO header	2
“b”	Buffer wire	1
“d”	Virtual LED	1

Table 2.16: String protocol interpretation for breadboard inputs and outputs.

If an “L” is read, the code reads the next character, which will either be a T or an F, to indicate a True or a False. The C++ code will store that Boolean value in a Boolean variable. If a “S” is read, the C++ code behavior is very similar to that of a power or ground plane; the code reads the next character, which will either be a T or an F, to indicate a True or a False. The C++ code will store that Boolean value in a Boolean variable. If a “g” is read, the C++ code will look at the next two characters that indicate the index for the GPIO vector array. The C++ code typecasts those next two characters into an integer and uses that as the index value to read or store new GPIO state Boolean values. If a “b” is read, the C++ code will look at the next value that indicates the index for the buffer vector array. Like the GPIO vector array, the C++ code typecasts the character into an integer, and uses that as the index value to be read or store new buffer state Boolean values. If a “d” is read, the code behavior is again similar to that of a buffer; the C++ code will look at the next character value, which describes the index value of the vector array, to either read or store new virtual LED Boolean states. Depending on the gate that was read prior (NOT gate, AND gate, etc.), the C++ code will then compute the output state based on what the input variable(s) are. The code will then store them in the proper output location, based on the string protocol.

If the C++ detects a change in the virtual LED states, which are captured and stored in the virtual LED state vector, the C++ will then concatenate each LED states into a string to be sent back to the JavaScript frontend as a sim2web polling message:
“led{led_number}={led_state}&led{led_number}={led_state}…”

Protocol Construction	Example
led{led_number}={led_state}&led{led_number}={led_state}…	led0=1&led1=0&led2=0&led3=0&led4=0

Table 2.17: String protocol construction for virtual LED states.

When the JavaScript receives this sim2web message, the JavaScript deconstructs the LED state. Any LED state that is a logic HIGH will change the virtual LED image to show one that is illuminated.

2.3.3 HARDWARE SCALABILITY

A primary goal of the RHL-Butterfly is the ability to easily scale the implementation to support many different target hardware. Although initial supported target hardware is the Intel DE1-SoC FPGA, the protocol interpretation through the backend Raspberry Pi Pico microcontroller was designed for the expansion of additional target hardware.

The DE1-SoC FPGA contains two General Purpose Input/Output ports (GPIO_0 and GPIO_1) with expansion headers that each total to 40 pins to be used to connect peripherals to the board.

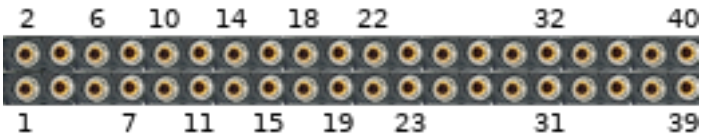

Figure 2.25: Virtual breadboard 40-pin GPIO header.

While some of the GPIO pins are reserved for other functionality for the Remote Lab, such as virtual DE1-SoC hardware switches and KEY buttons, any remaining GPIO pins can be used to directly interface with virtual external peripherals, such as the virtual breadboard.

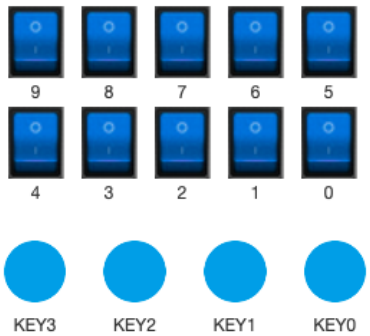

Figure 2.26: LabsLand virtual FPGA switches and KEYs.

Different target hardware microcontrollers offer a different number of GPIO pins for their expansion headers, thereby increasing or decreasing the number of available input and output GPIO pins to interface with virtual peripherals such as the virtual breadboard. For example, an Arduino UNO has two single row expansion headers, totaling to 32 total pins for both combined expansion headers, which is less than a single expansion header for a DE1-SoC.

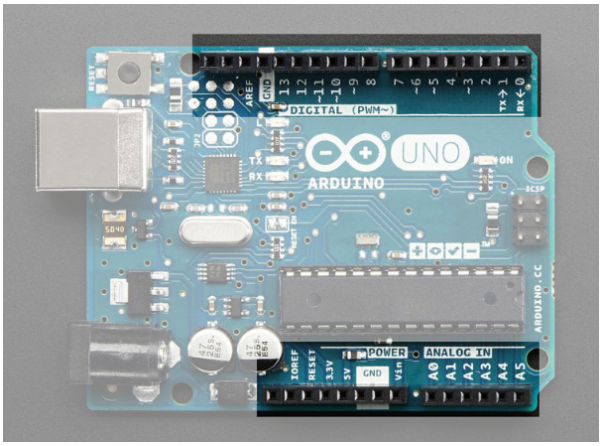

Figure 2.27: Arduino UNO GPIO header. [21]

For any target hardware, however, the underlying C++ functionality remains the same: the code obtains the same string protocol from the JavaScript frontend, to which it then parses and tracks the GPIO states, buffer states, and virtual LED states with C++ vectors. Different target hardware may have different number of available input and output GPIO due to a different header size available and pinouts for the microcontroller. Because the functionality of the backend microcontroller remains the same, it provides an opportunity the support for expandability within the C++ code.

The C++ update function subsides within a Breadboard class called `ButterflySimulation`, to allow for code reusability on different projects. A secondary class, `DE1SoC_BufferflySimulation`, inherits the `ButterflySimulation` parent class. The child class contains solely two functions: a `getNumberOfSimulationInputs`, which returns the number of available GPIO inputs based on that target hardware, and a `getNumberOfSimulationOutputs`, which returns the number of available GPIO outputs based on that target hardware. All other functions are inherited from the parent class. The number of simulation inputs and simulation outputs dictate the vector array for the output GPIO tracker and input GPIO tracker, which maps to the target hardware physical GPIO. As more

target hardware become supported with the virtual breadboard, additional child classes will be added that inherits `BufferflySimulation`, each returning a different number from `getNumberOfSimulationInputs` and `getNumberOfSimulationOutputs`.

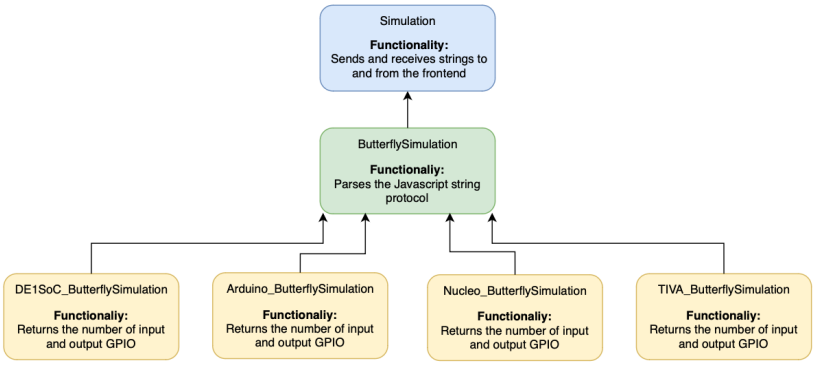

Figure 2.28: Raspberry Pi Pico class inheritance.

2.3.4 Software Scalability

At its launch, the RHL-Butterfly was one of two available plugins that interfaced with the open-source LabsLand simulation libraries, designed to give students a virtual breadboard experience that interfaces with the GPIO of the target hardware. The second plugin for the LabsLand simulation library was a completely virtualized, digital twinning version of a 3D parking lot whose frontend was constructed using the BabylonJS infrastructure and communicated with the Raspberry Pi Pico using POST and Polling networking methods.

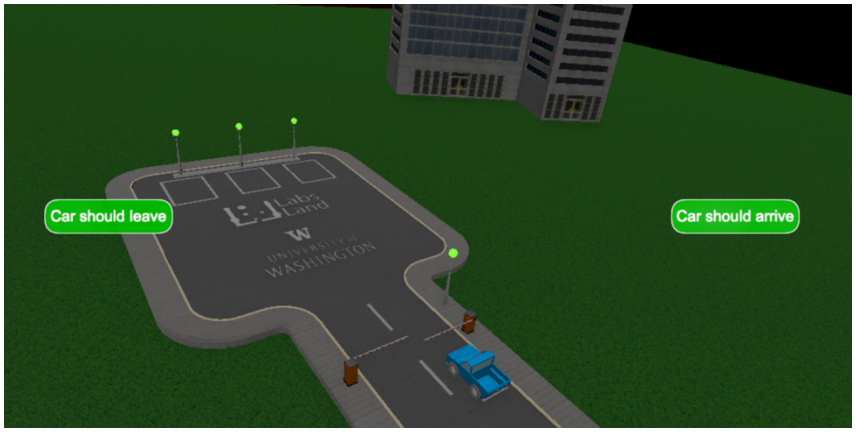

Figure 2.29: 3D Parking Lot user interface.

The digital twinning 3D parking lot provides two UI element buttons, one for adding a car to the parking lot, and another to force a car to leave that parking lot. That information, when pressed,

acts similarly to that of the RHL-Butterfly virtual breadboard's user submit button, to which a string protocol is then created and sent over through POST to the Raspberry Pi Pico. The digital twinning 3D parking lot also contains four LEDs that can switch between the red and green color. These states are similar to the virtual LED found in RHL-Butterfly; the states are tracked through a vector array in the shared memory of the Raspberry Pi Pico. At the end of an update cycle, the network uses Polling to send the virtual LED color states back to the JavaScript frontend, to which the UI changes the virtual LED states accordingly.

With the goal of scalability in mind, any additional interface can be constructed using similar approaches: a JavaScript frontend that provides the generalized UI for the students or user to interact virtually with the target hardware, a POST method that deconstructs the UI element into strings of information to be passed to a Raspberry Pi Pico, and a C++ firmware running on the embedded system that synchronously processes the strings of information passed and updates the virtual state of the UI element.

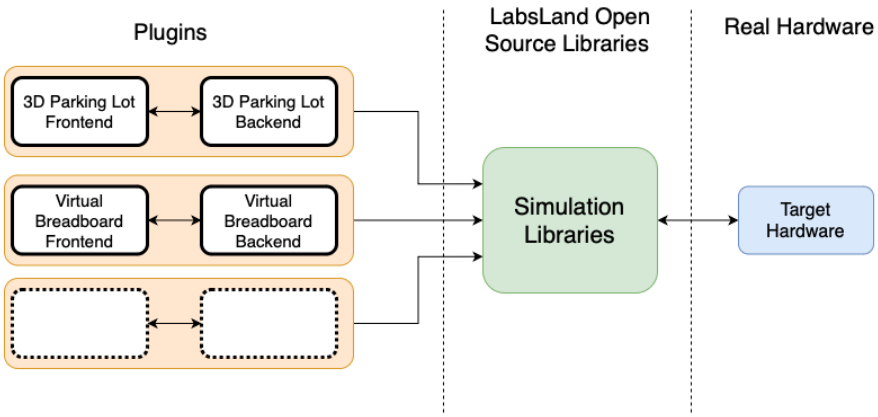

Figure 2.30: Remote laboratory simulation plugin topology.

From there, the open-source LabsLand Simulation libraries updates the necessary GPIO of the target hardware to match the virtual interface provided from the JavaScript UI.

Chapter 3. RESULTS

3.1 UPDATE GRAPHS

The frontend virtual breadboard and the backend microcontroller processing unit communicates using bidirectional communication. Whenever a new breadboard design is created, the frontend checks for circuit errors. If there are none, the JavaScript then parses the designed circuit into a minimized string representation and sends it to the Raspberry Pi Pico using a REST POST method.

The microcontroller, running independently from the virtual breadboard client, periodically calls an `update()` function that computes the last sent string protocol from the JavaScript. Even when no new virtual breadboard circuit information is created, the Raspberry Pi Pico will continue to periodically compute the last information sent. The benefit of this is twofold: first, by periodically calling the `update()` function, it ensures that the computation remains the most up-to-date version, and second, the behavior then mimics that of physical hardware through a period of time that contains metastability, or a small period of time where the final output of a chain of digital circuits has not yet saturated. These behaviors are shown in the example circuits constructed below.

3.1.1 CIRCUIT 1

The first example constructed uses two virtual SPDT switches to control the input logic level of different logic gates. An OR logic gate has one input connected to a virtual switch, with the other input connected to GND. The output of the OR gate is then fed as input one of an AND gate, with the other input connected to the second virtual switch. The output of the AND gate is then connected to an input of a NOT gate, whose output is connected to another NOT gate. The output of the second NOT gate is connected to both a virtual LED and an output GPIO.

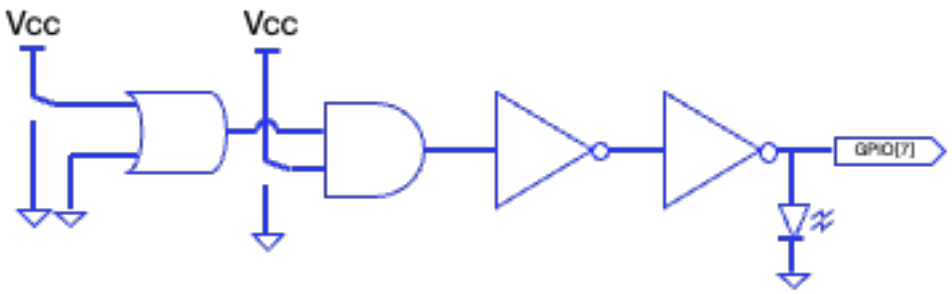

Figure 3.1: Digital circuit for circuit 1.

When constructed on the RHL-Butterfly virtual breadboard, the constructed JavaScript string protocol is “nb1b2;nb2g00,d0;ab0STb1;oSTLFb0;\n”.

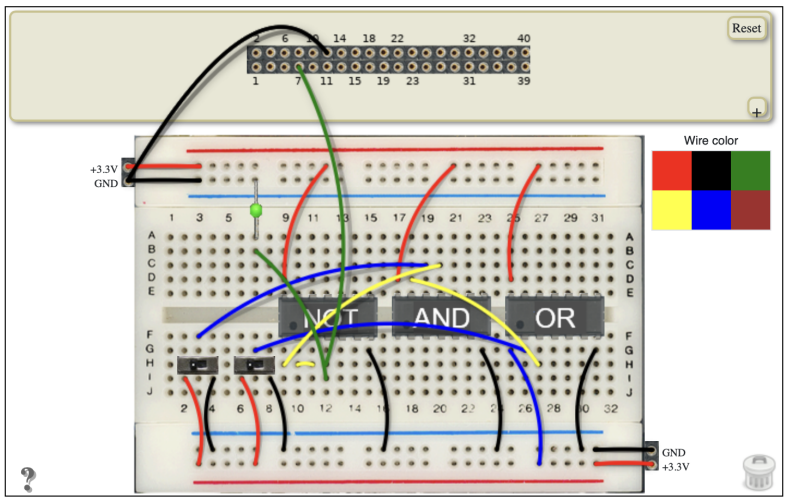

Figure 3.2: Virtual breadboard construction for circuit 1.

In this circuit, three total intermediate buffers are used to create the constructed circuit. For each update call, the Raspberry Pi Pico computes the buffer states one time, and stores the computed states into a vector array in shared memory, which is originally initialized as 0. Upon the next iteration, the computation uses the buffer output digital logic states that were computed in the prior iteration to re-compute each state, thereby updating the digital logic states for the buffers.

	Iter 1	**Iter 2**	**Iter 3**	**Iter 4**	**Iter 5**	**Iter 6**	**Iter 7**	**Iter 8**	**Iter 9**	**Iter 10**	**Iter 11**
GPIO 7	0	0	0	1	1	1	1	1	1	1	1
LED 0	0	0	0	1	1	1	1	1	1	1	1
Buffer 0	0	1	1	1	1	1	1	1	1	1	1
Buffer 1	0	0	1	1	1	1	1	1	1	1	1

Buffer 2	0	1	1	0	0	0	0	0	0	0	0
Buffer 3	0	0	0	0	0	0	0	0	0	0	0
Buffer 4	0	0	0	0	0	0	0	0	0	0	0

Table 3.1: Buffer, LED, and GPIO digital states for multiple update iterations.

When the buffer states and associated GPIO and LED outputs are still digitally volatile, the computation behaves as metastability would when using real-world hardware. At some point, the output and buffer states saturate into the final buffer and output states. For this circuit, the buffers and outputs saturate after the 4th iteration of the `update()` call.

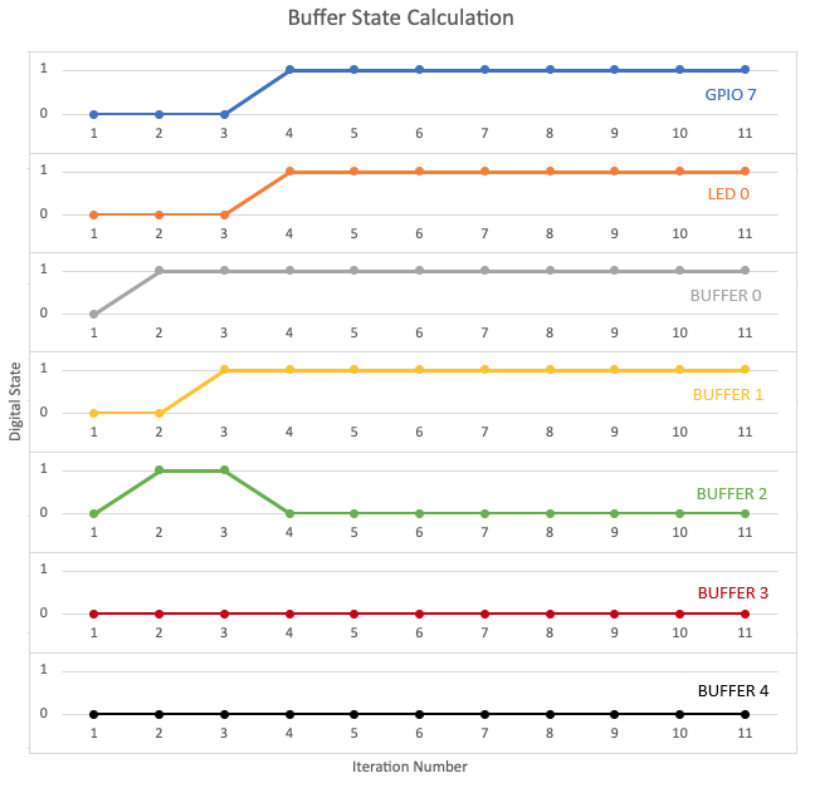

Table 3.2: Table representation of the digital states for multiple update iterations.

3.1.2 CIRCUIT 2

The next example constructs two independent digital logic circuits and uses two distinct logic gates: OR gates and NOT gates, whose outputs are connected to different GPIO outputs. In one digital logic circuit, a virtual switch is used as an input to an OR gate, with the other input connected to GND. The output of that OR gate is connected to a GPIO output. For the second digital logic circuit, a virtual switch is used as an input to the OR gate, with the other input connected to an input GPIO. The output of the OR gate is connected to an input of a NOT gate, whose output is connected to an input of another OR gate. The second input of the OR gate is connected to GND. This process is then repeated: the output of the OR gate is connected to an input of another NOT gate, whose output is connected to an input of an OR gate. This time, the second input is connected to a virtual switch. The output of the OR gate is then inverted through a NOT gate before being connected to an output GPIO.

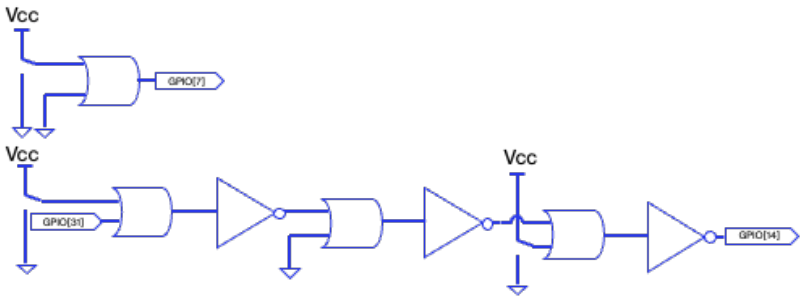

Figure 3.3: Digital circuit for circuit 2.

When constructed on the RHL-Butterfly virtual breadboard, the constructed JavaScript string protocol is “nb2b3;nb4g02;nb0b1;oSTg00b0;ob3STb4;ob1LFb2;oLFSTg00;\n”.

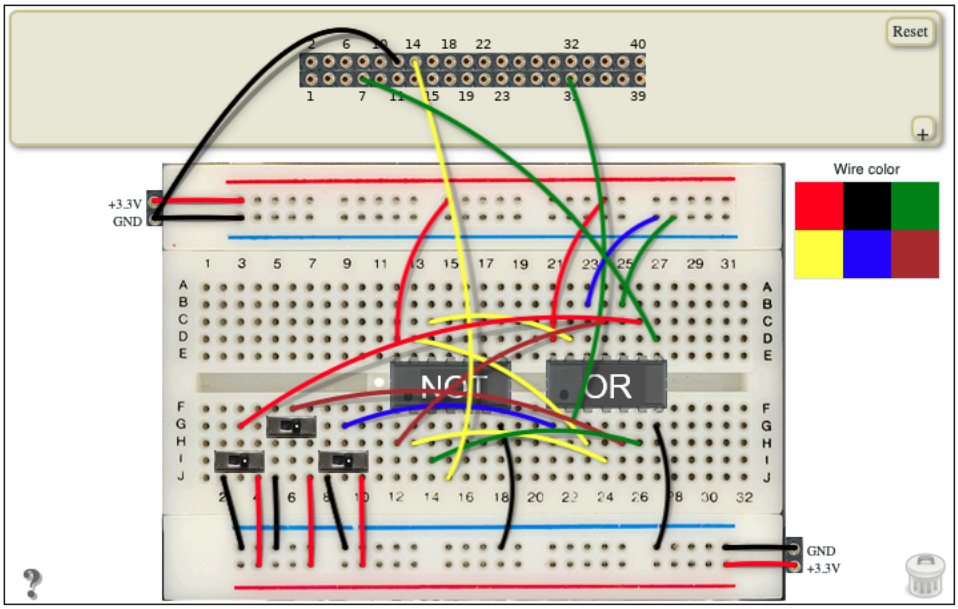

Figure 3.4: Virtual breadboard construction for circuit 2.

In this circuit, five intermediate buffers are used. Similar to as before, for each update call, the Raspberry Pi Pico computes the buffer states one time, and stores the computed states into a vector array in shared memory, which is originally initialized as 0. Upon the next iteration, the computation uses the buffer output digital logic states that were computed in the prior iteration to re-compute each state, thereby updating the digital logic states for the buffers.

	Iter 1	**Iter 2**	**Iter 3**	**Iter 4**	**Iter 5**	**Iter 6**	**Iter 7**	**Iter 8**	**Iter 9**	**Iter 10**	**Iter 11**
GPIO 7	0	1	1	1	1	1	1	1	1	1	1
GPIO 14	0	1	0	0	0	0	0	0	0	0	0
Buffer 0	0	1	1	1	1	1	1	1	1	1	1
Buffer 1	0	1	0	0	0	0	0	0	0	0	0

Buffer 2	0	1	0	0	0	0	0	0	0	0	0
Buffer 3	0	1	0	1	1	1	1	1	1	1	1
Buffer 4	0	1	1	1	1	1	1	1	1	1	1

Table 3.3: Buffer and GPIO digital states for multiple update iterations.

For this circuit, the buffers and outputs completely saturate after the 4th iteration of the `update()` call.

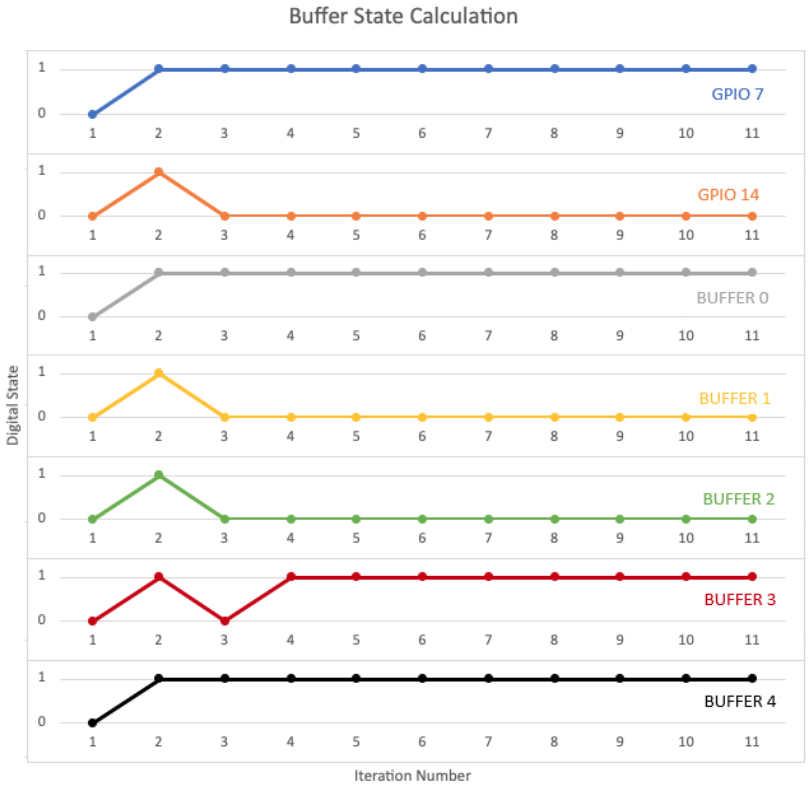

Figure 3.5: Table representation of the digital states for multiple update iterations.

3.1.3 CIRCUIT 3

In some circuits, saturation cannot be reached due to an intrinsic oscillation from the created circuit. An example of a circuit that exhibits this behavior is shown in this example. Three inverters are chained together. For the output of one NOT gate, a virtual LED is connected.

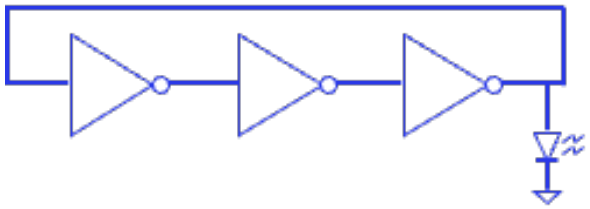

Figure 3.6: Digital circuit for circuit 3.

When constructed on the RHL-Butterfly virtual breadboard, the constructed JavaScript string protocol is “nb2b0,d0;nb0b1;nb1b2;\n”.

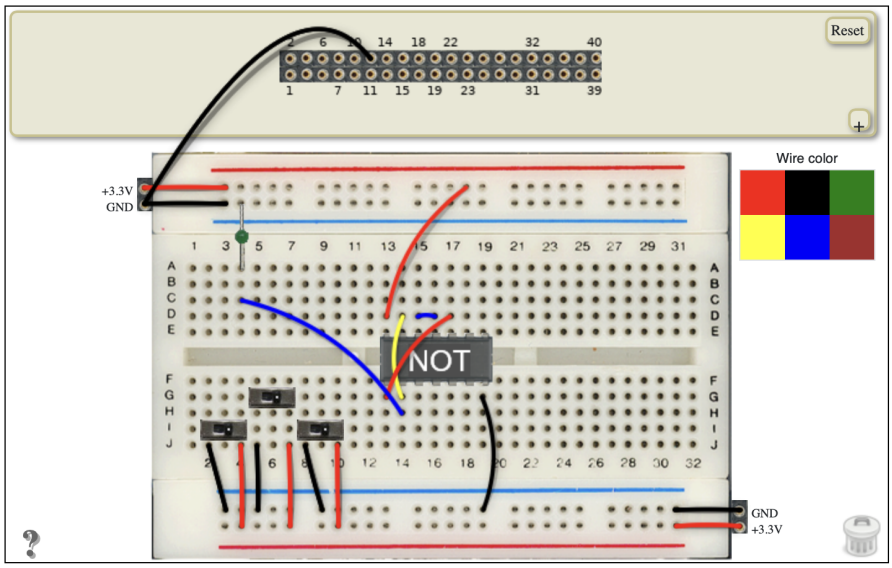

Figure 3.7: Virtual breadboard construction for circuit 3.

If an intrinsic oscillation occurs, certain buffer indices will force a race condition from the buffer state calculation in an attempt to update the buffer state according to its desired digital logic output. When this occurs, each iteration of the `update()` function parses the string protocol blocks at a time from left to right, where each block is separated by the “;”. Therefore, after each `update()`, the last block in the string protocol takes the highest precedence due to it being the last block to be processed in the `update()` function.

	Iter 1	**Iter 2**	**Iter 3**	**Iter 4**	**Iter 5**	**Iter 6**	**Iter 7**	**Iter 8**	**Iter 9**	**Iter 10**	**Iter 11**
LED 0	0	1	0	1	0	1	0	1	0	1	0
Buffer 0	0	1	0	1	0	1	0	1	0	1	0
Buffer 1	0	0	1	0	1	0	1	0	1	0	1
Buffer 2	0	1	0	1	0	1	0	1	0	1	0
Buffer 3	0	0	0	0	0	0	0	0	0	0	0
Buffer 4	0	0	0	0	0	0	0	0	0	0	0

Table 3.4: Buffer, LED, and GPIO digital states for multiple update iterations.

The speed of the oscillation is thereby determined by the time of computation for the `update()` function, as well as the timing interval between each `update()` function call.

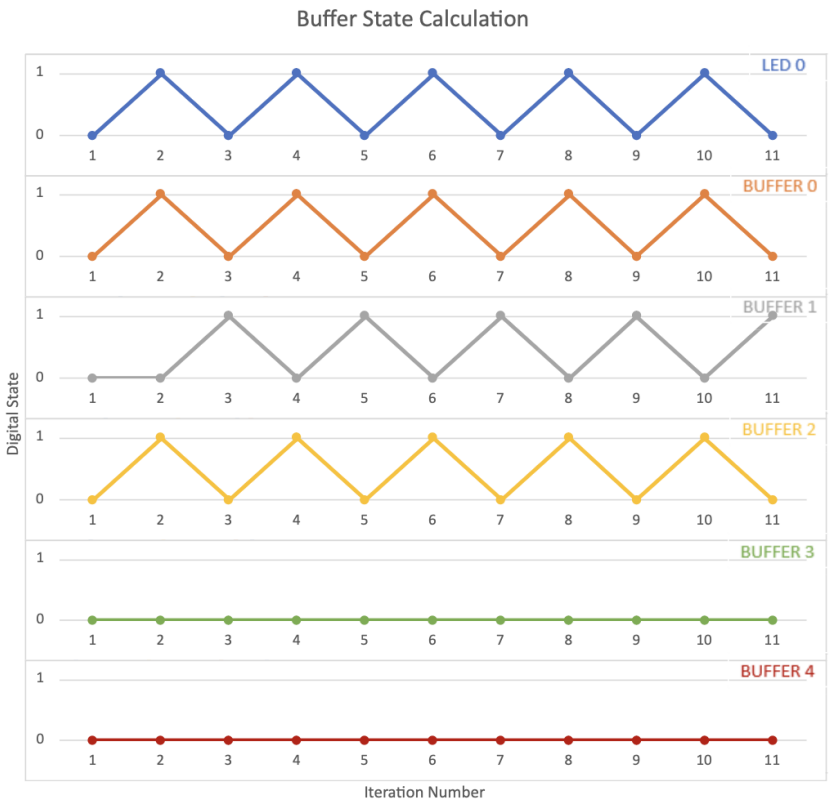

Figure 3.8: Table representation of the digital states for multiple update iterations.

3.2 LabsLand Integration

RHL-Butterfly is hosted on the LabsLand network to establish bidirectional communication between the target hardware (i.e. DE1-SoC) and the frontend virtual breadboard GUI. When a user compiles and synthesizes their code for a microprocessor, they can upload their code to one of the available hardware platforms available in the lab, to which the LabsLand website redirects the user to the virtual laboratory interface, consisting of a livestream of the actual hardware, virtual pushbuttons and switches, and the virtual breadboard to interface the hardware GPIO. To minimize a queue wait time for remotely accessing the target hardware, a three-minute time limit is strategically set in place for the user to demonstrate their design and ensure proper functionality.

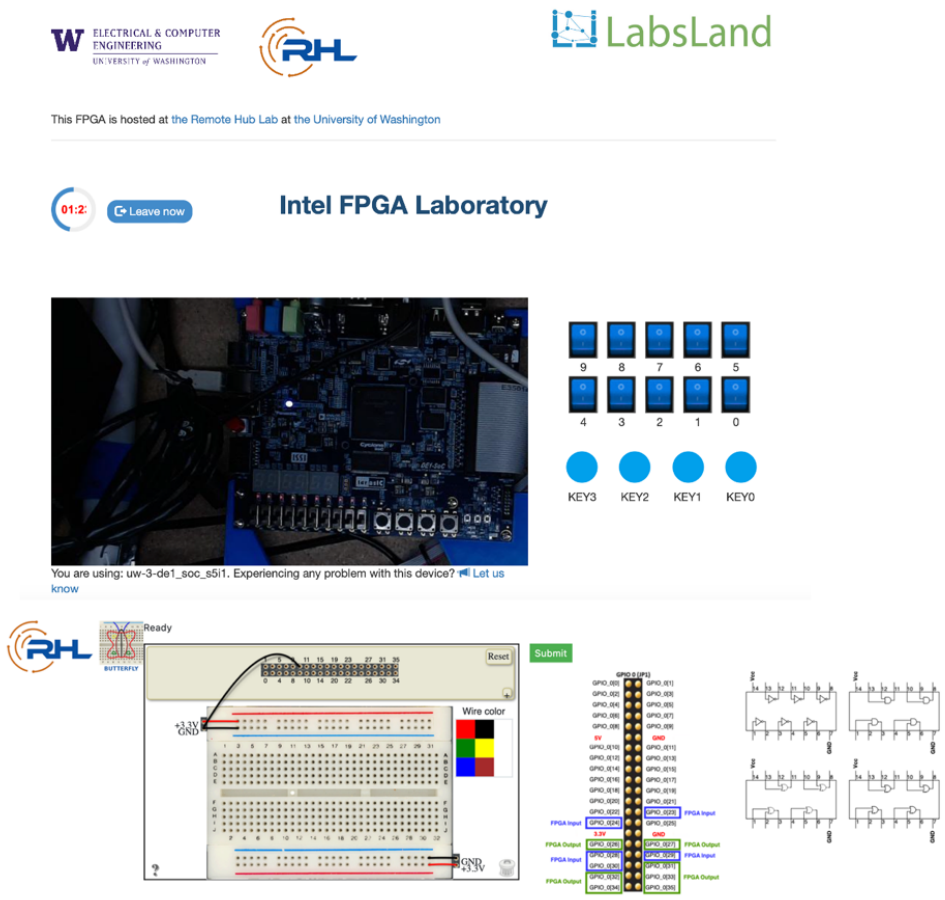

Figure 3.9: Full virtual breadboard LabsLand user interface.

Students initially expressed concerns about synchronously constructing their breadboard circuit within the three-minute time frame, based on the feedback results below.

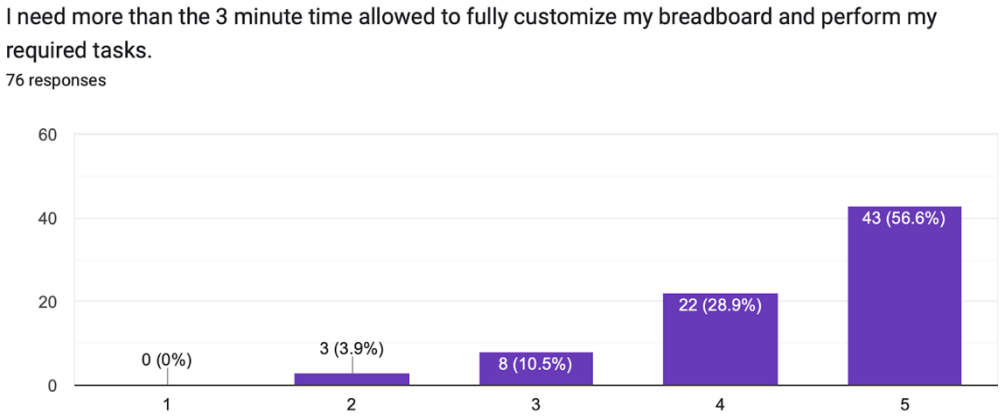

Figure 3.10: Survey results for 3 minute time limit.

To combat this, an additional UI element for the virtual breadboard was constructed that allows an asynchronous way to construct their breadboard designs within the LabsLand IDE platform. Students' breadboard configurations through the LabsLand IDE platform are then saved and restored for when the students are accessing the remote hardware with their compiled code. This workflow not only provides additional time for the students to carefully experiment and construct their breadboard, but provides a workflow that mimics that of a physical laboratory: allowing students to build their circuits in parallel with writing their SystemVerilog or embedded programming designs.

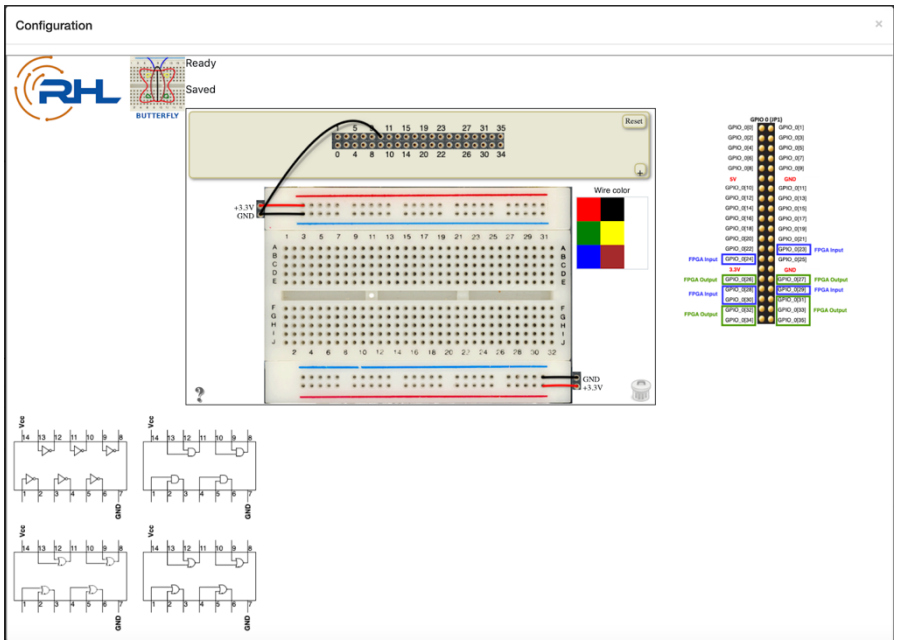

Figure 3.11: User interface element for asynchronous breadboard experimentation.

3.3 Integration to Curriculum

The new iteration of the virtual breadboard was deployed and offered in two academic terms of the University of Washington Design of Digital Circuits course that used the remote FPGA lab. These two academic terms were offered in 2023 as in-person course offerings, after the remote course offerings from the COVID-19 pandemic. In both of these academic terms, students used the new generation virtual breadboard to construct breadboard designs featuring virtual SPDT breadboard switches and LEDs, to interface with the GPIO of the DE1-SoC for their laboratory assignment. The objective of the laboratory assignment was to review students' knowledge on Mealy and Moore Finite State Machines learned from prior, prerequisite courses, through a system that detects the entering and exiting of a car in a parking lot. Using two virtual SPDT switches, their SystemVerilog implementation must recognize the following pattern to increment the number of cars in a given parking lot, and the following pattern to decrement the number of cars in that given parking lot.

Increment: `{inner_sensor, outer_sensor} = 00 → 01 → 11 → 10 → 00`

Decrement: `{inner_sensor, outer_sensor} = 00 → 10 → 11 → 01 → 00`

Completing the laboratory assignment requires several steps. First and foremost, students must have a thorough understanding of the GPIO mapping on the FPGA for the GPIO they will be working with. This knowledge is crucial for constructing a proper working circuit on the virtual

breadboard, which is the second step in the assignment. Once the virtual breadboard is properly constructed, students must interface with its inputs using their SystemVerilog FPGA implementation. This will enable them to keep track of a parking lot occupancy counter using the Mealy and Moore Finite State Machine concepts learned from prerequisite courses. Overall, the successful completion of this assignment requires a strong understanding of both hardware and HDL concepts.

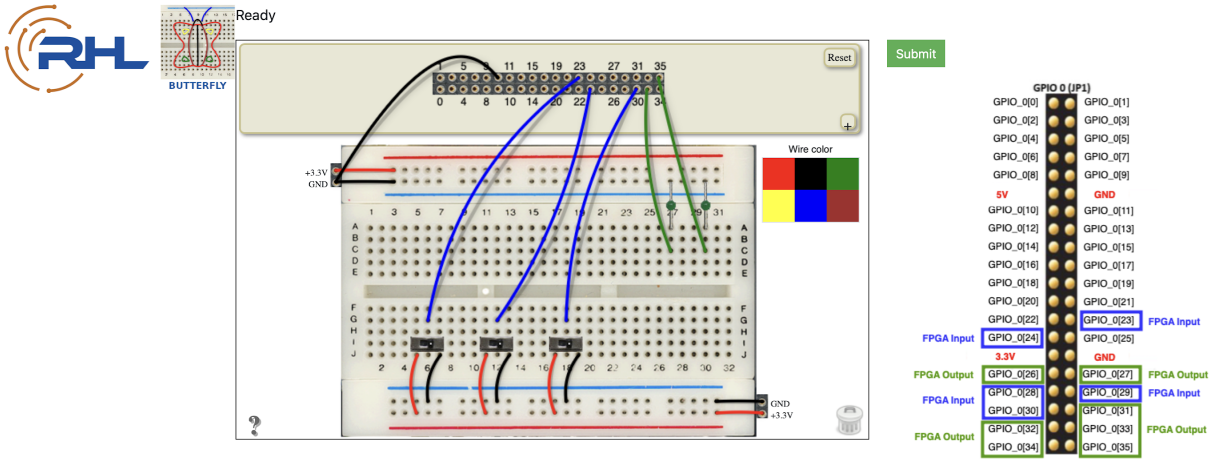

Figure 3.12: Example configuration of the virtual breadboard for Design for Digital Circuits and Systems.

3.3.1 Survey Construction

After completing their lab assignments featuring the improved virtual breadboard version, students were asked to participate in a completely anonymous and optional usability study through a survey. The goal of the survey for the virtualized breadboard was to evaluate student perspectives on the viability of a remote breadboard learning experience over a traditional physical breadboard. The survey questions were broken down into four distinct categories: the individual features and components of the virtual breadboard, the virtual breadboard's ability to interface with the DE1-SoC FPGA, the equitable access nature of the virtual breadboard, and the overall virtual breadboard experience. The individual features of the virtual breadboard category provide insight on the frontend design, which include the helpfulness of the error messages, the usability of drawing wires, and the ability to drag the breadboard components, each with the intention of identifying key feature shortcomings that can be upgraded in future iterations of the virtual breadboard. The ability to interface with the target hardware category is used to characterize the effectiveness of the backend design. The virtual breadboard equitable access

nature category is used to further explore the potential long term usage of remote laboratories for engineering curriculums, even in a post pandemic world.

Question Category	**Question Phrasing**	**Question Type**
Overall Virtual Breadboard Experience	The virtual breadboard is intuitive.	Likert Scale
	Do you have prior physical breadboard experience?	Multiple Choice
	The virtual breadboard is similar to that of a physical breadboard.	Likert Scale
Virtual Breadboard Equitable Access	Where did you spend most of the time completing the lab assignment?	Multiple Choice
	It is convenient to access this virtual breadboard at any location with an internet connection.	Likert Scale
	It is convenient to access this virtual breadboard at any time that fits my schedule.	Likert Scale
Individual Features	Dragging components in this virtual breadboard has similar intuition as component placement in a physical breadboard.	Likert Scale
	The ability to drag components helped in my understanding of how breadboards work.	Likert Scale
	The error messages displayed (if any) were helpful in debugging my circuit.	Likert Scale
	The ability to draw wires was helpful in my breadboard experimentation.	Likert Scale
	The ability to draw differently colored wires was helpful.	Likert Scale
Virtual Breadboard GPIO	Interfacing the virtual breadboard's GPIO was similar to that of a physical breadboard.	Likert Scale
	The virtual breadboard helped me understand the FPGA GPIO interface.	Likert Scale
Virtual Breadboard Suggestions	Additional helpful aspects of the breadboard.	Free response
	Suggestions for future improvements.	Free response

Table 3.5: Survey breakdown for the virtual breadboard.

Many of the questions were phrased in a positive way and had answer choices on a 5-point Likert scale, where 1 indicated that the students strongly disagreed with the question statement, and a 5 indicated that the students strongly agreed with the question statement.

Likert Scale	Meaning
1	Strongly Disagree
2	Disagree
3	Neutral
4	Agree
5	Strongly Agree

Table 3.6: Likert scale associated meanings for survey quantitative analysis.

3.3.2 Survey Results per Category

After the lab assignment, students were asked to participate in an anonymous survey relaying their experiences with the virtual breadboard. For each of the two academic terms, the surveys given were similar, though not identical. The virtual breadboard continued to be mutable to gain better usability, and as such, the survey was updated between the two academic terms to reflect the current state of the virtual breadboard. For the first academic term that the virtual breadboard was deployed in, referred to as Academic Term 1 for the remainder of the paper, out of 92 distinct groups, 76 unique responses were gathered for a response rate of 83%. For the second academic term, referred to as Academic Term 2, out of 55 distinct groups, 34 unique responses were gathered for a response rate of approximately 62%.

3.3.2.1 Overall Breadboard Experience

Students were initially asked to rate on a 5-point Likert scale how intuitive the virtual breadboard was to use, where a 5 indicated that the breadboard was very intuitive, and a 1 indicated that the virtual breadboard was very unintuitive. This question provides students the opportunity to think about the entire experience of the virtual breadboard, and was intentionally asked near the beginning of the survey to avoid having the students narrow their focus on one particular feature, which later sections of the survey addresses.

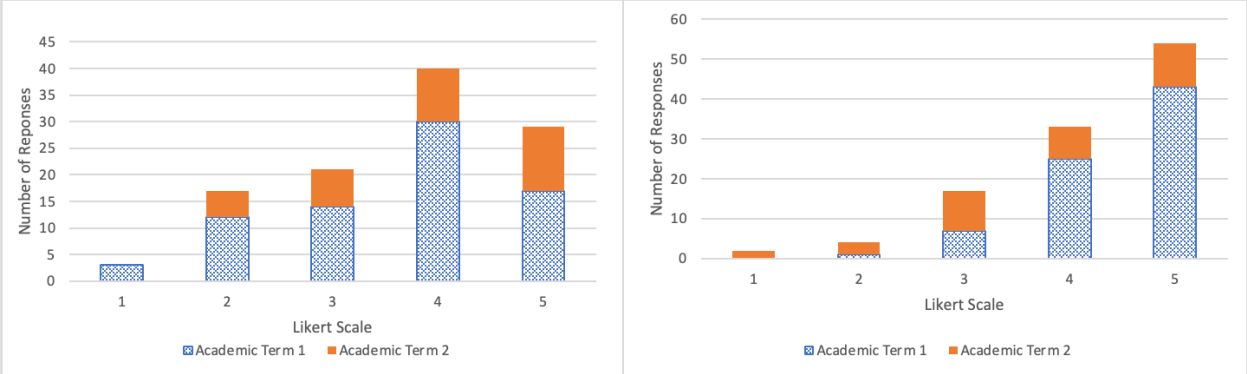

Figure 3.13: Student agreement that a) the virtual breadboard was intuitive to use (left), and b) the virtual breadboard felt similar to that of a traditional physical breadboard (right).

The responses in Figure 3.13a are right skewed, showing that the majority of the students agreed with the statement that the virtual breadboard was intuitive to use. It is worth noting, however, that many students have had prior experience with physical breadboards from previous prerequisite courses, thus the familiarity with the physical breadboard counterparts likely added to the relative ease of virtual breadboard intuition, as shown in Figure 3.14.

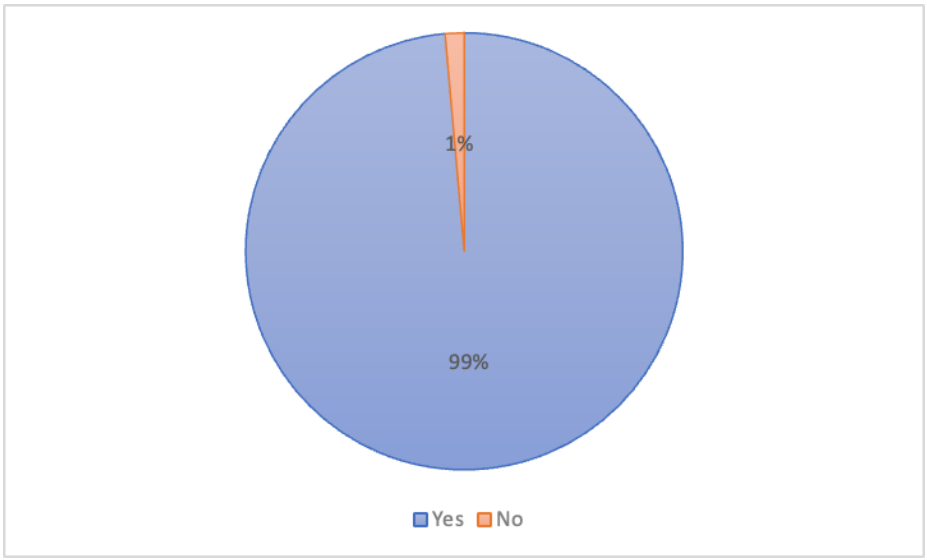

Figure 3.14: Amount of responded students that had prior experience with physical breadboards.

The next question added a quantitative metric to show the similarities between a physical breadboard to the provided virtual one, where students were asked to rank on the same 5-point

Likert scale how similar the virtual breadboard experience was as a whole to their prior breadboard experience. The responses shown in Figure 3.13b shows that many students agreed that their experience with using the virtual breadboard was very similar to that of their prior experience with physical breadboards.

Sample size for students with no prior physical breadboard experience (N = 1) is too small to make a justifiable conclusion that initial learning through a virtual breadboard can adequately prepare students for physical breadboard counterparts later on through their education and/or career; only a conclusion of physical breadboard aiding in the understanding of a virtual one can be obtained. However, given the promising results showing the intuitive nature of a virtual breadboard and its similarity with a physical breadboard, the virtual breadboard platform delivers a promising approach for a viable alternative to physical breadboards.

3.3.2.2 Virtual Breadboard Individual Features

The next section specifically addresses individual features of the virtual breadboard that students encountered with integration to their FPGA laboratory assignments, and their similarities and viable replacements to physical breadboards.

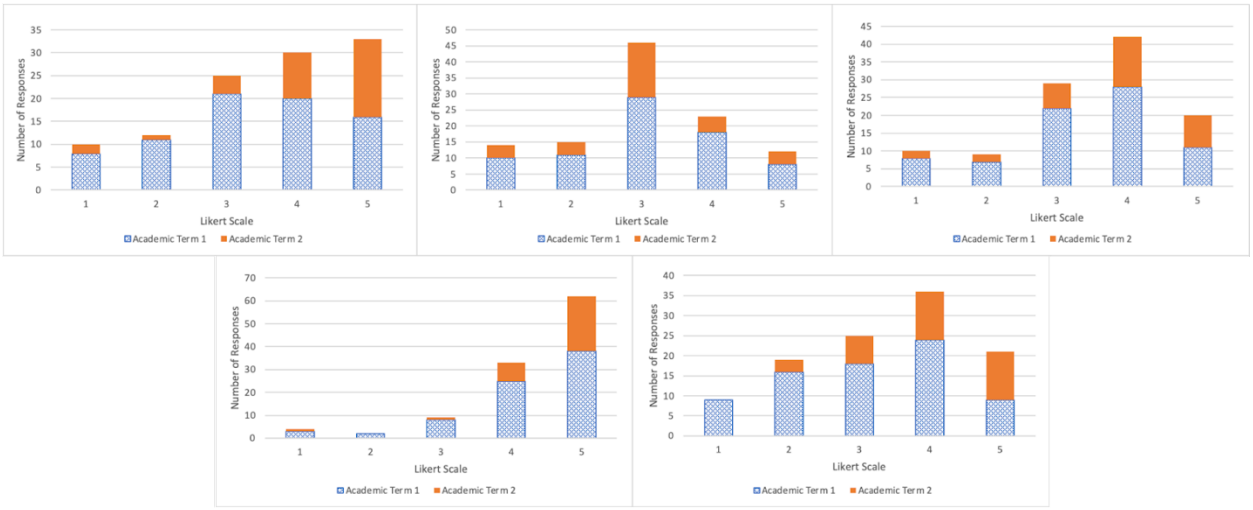

Figure 3.15: Students' agreement to a) ability to drag components around on the virtual breadboard felt intuitive (top-left), b) dragging components helped with their ability to learn about breadboards (top-middle), c) the ability to draw wires was useful in their experimentation (top-right), d) the ability to draw differently colored wires was useful (bottom-left), and e) the error messages was helpful in debugging their designs (bottom-right).

When asked if the ability to drag the components anywhere on the virtual breadboard (a differentiating feature from the previous virtual breadboard experience) provided a similar intuitive nature as that of a physical breadboard, the responses were a little more mixed, particularly in Academic Term 1, indicated in Figure 3.15a, which showed a relatively flat response across the 5-point Likert scale. Figure 3.15a also shows that student responses contain a large difference between Academic Term 1 and Academic Term 2. Differences between the deployment for the RHL-Butterfly between these two terms is that the second term allows students to asynchronously create their breadboard prior to demonstrating their SystemVerilog designs, relieving some of the time constraints and pressure. By doing this, students may be going about a more methodical approach for their breadboard design construction.

When asked if the draggability feature implemented to freely place the breadboard components helped in the students' understanding of how breadboards should work and helped in facilitating the knowledge of breadboard electrical nodes and characteristics (Figure 3.15b), the responses again were relatively mixed, with the majority of students indicating a neutral response.

Using thematic analysis on a later question about the virtual breadboard design suggestions for improvements, as shown in Table 3.8, to which the students were asked to write improvement ideas on the virtual breadboard for future iterations, a clear trend arises that better describes the reasons for the mixed results of the draggability feature. Here, a byproduct of having components be draggable to different locations of the virtual breadboard are accidental drags, particularly with components that require many user interactions, such as the SPDT switches. Students believe a way to incorporate the draggability feature without accidental drags would be to have a way that saves the breadboard state and locks the components in place on the virtual breadboard. This is a feature that can be added for future iterations of the virtual breadboard.

Students were asked if the debug errors were useful in helping them construct their breadboard designs, shown in Figure 3.15e. The results for this question also received relatively flat responses but is due to the wide variety of prior breadboard experiences; some students may have relied on the debug messages, while others may not have experienced the messages at all due to an initially correct constructed circuit.

The virtual breadboard allows for the ability to draw intermediate wires on any node for the breadboard, a similar experience to adding jumper cables and wires on a physical breadboard. This is to promote different ways of creating electrical circuits and breadboard designs. When asked if this ability to draw wires has helped the students with breadboard experimentation, as shown in Figure 3.15c, using the 5-point Likert scale, students responded positively, and agreed that this feature has helped with their breadboard experimentation.

Following the question on whether drawing intermediate wires was useful, support to draw differently colored wires is also a feature in the RHL-Butterfly in an attempt to aid students in color-coding their designs. When asked whether the differently colored wires were a feature that proved to be useful in their breadboard construction, in Figure 3.15d, students overwhelmingly agreed.

3.3.2.3 Virtual Breadboard GPIO

A main goal of using the virtual breadboard is to help students learn about interfacing with the DE1-SoC GPIO to connect external components and integrated circuits to their FPGA designs. The core motivation for this section is to check if the virtual GPIO interface with physical DE1-SoC provided a similar experience with that of a physical GPIO interface, and if the virtual breadboard implementation of that interface had aided in students' knowledge in accessing the GPIO through their SystemVerilog implementation.

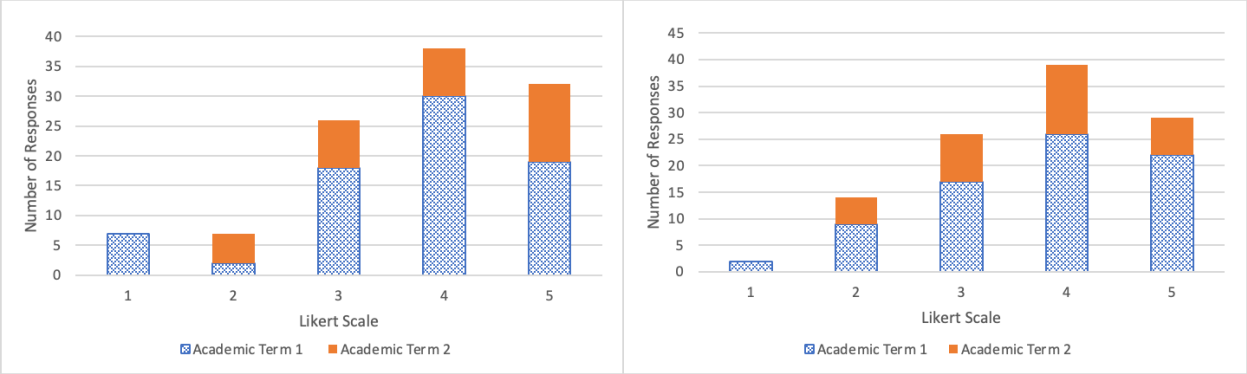

Figure 3.16: Students' agreement to a) the virtual breadboard helped in their understanding of interfacing with the FPGA GPIO (left), and b) the interfacing nature of the virtual breadboard felt similar to that of a traditional physical GPIO interface (right).

Students were initially asked to rate whether interfacing with the DE1-SoC GPIO on the virtual breadboard was similar to that of a physical breadboard and physical FPGA on a 5-point Likert scale, in Figure 3.16a, to which many students agreed, with the results skewed to the right. Students were then asked to rate on a 5-point Likert scale whether using the virtual GPIO had helped in their understanding of how to interface with DE1-SoC GPIOs, both through hardware connections and through their SystemVerilog implementation, indicated in Figure 3.16b. A majority of students with a 4 and 5, indicating they believed the virtual breadboard had helped them understand interfacing with GPIOs.

3.3.2.4 Equitable Access

The next three questions specifically address the equitable access and the virtual nature of the remote laboratory experience, specifically regarding the virtual breadboard implemented in their lab.

The virtual breadboard remote aspect survey theme was only asked in Academic Term 1, in large part due to differences in teaching styles for the two different terms. The instructor for Academic Term 2 highly recommended doing labs during in-person Teaching Assistant Office Hours on campus, and had designated lecture work times to work on their labs. The first term instruction offered a larger amount of location flexibility for students with available Zoom office hours from Teaching Assistants.

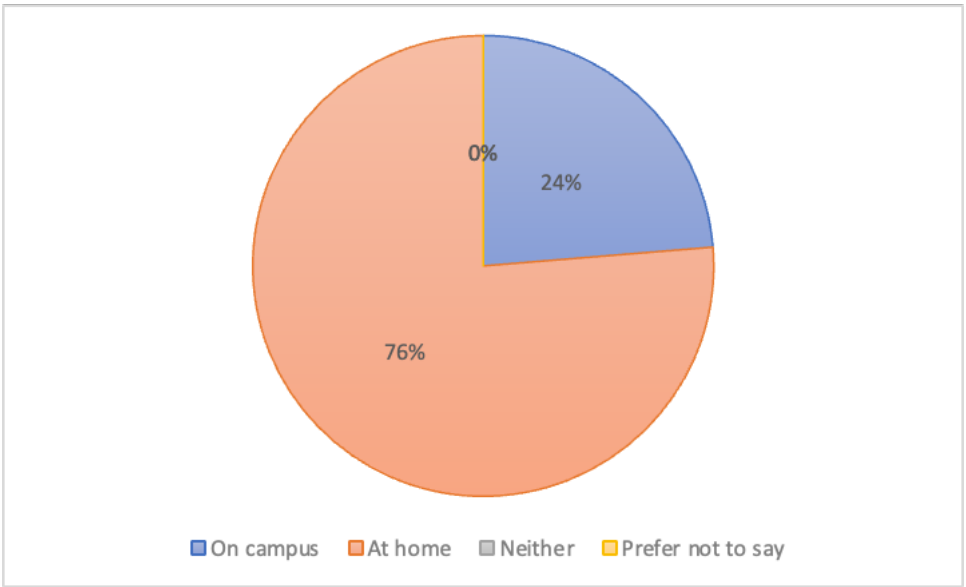

Figure 3.17: Students' responses to where they indicated they spent the most time working on their lab assignment.

The first question within this survey category asked students, if they were comfortable with sharing, where they spent the majority of their time completing their laboratory assignment. An overwhelming majority of students responded that they had spent the most amount of time at home, indicated in Figure 3.17.

Additionally, when asked to rate the convenience of accessing the virtual breadboard online and allowed location flexibility, an overwhelming majority of students responded with a "5", indicating that the location flexibility was strongly convenient.

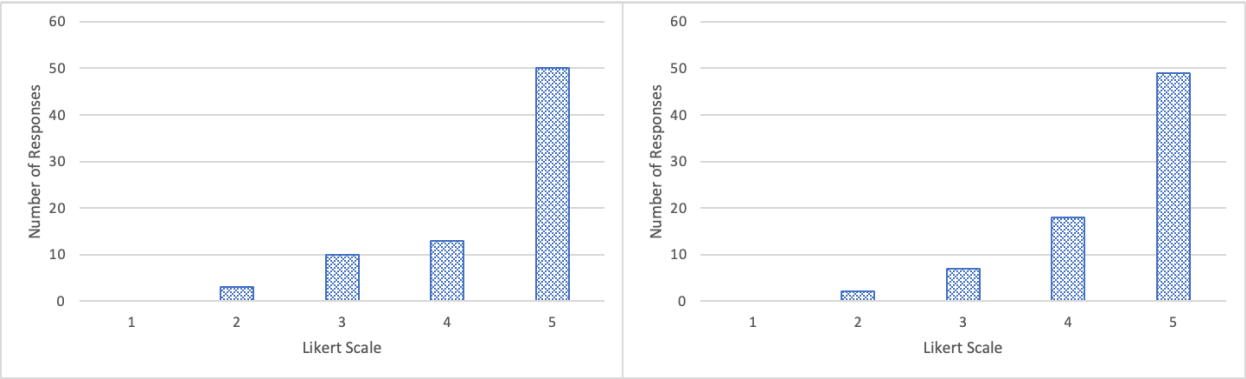

Figure 3.18: Students' agreement to a) the virtual breadboard's location flexibility (left), and b) the virtual breadboard's time flexibility (right).

A similar question was then asked about the convenience of accessing the virtual breadboard at any time that fit their desired homework schedule. Again, an overwhelming majority of students responded with a "5", indicating that the time flexibility was strongly convenient. Overall, students responded very positively to the notion of greater flexibility through remote laboratories and remote learning for virtual breadboard experimentation.

3.3.2.5 Free Response Themes

Near the end of the survey, students were provided with an opportunity to describe in free-response format additional helpful aspects of the virtual breadboard as well as suggestions for future improvements. A thematic analysis was conducted on both free-response answers to identify common themes and characteristics from the student population.

Free Response Question	**Keywords and Phrases**	**Inferred Theme**
Helpful aspects of the virtual breadboard	Error messages, debugging messages, error codes, connection error checks, debugging, error-checks	Breadboard error message features
	Free, location flexibility, don't need physical breadboard, working remotely	Equitable virtual breadboard access
	Visibility, dragging components, GPIO diagram nearby, clean interface, changing wire angles	Virtual breadboard interface

Table 3.7: Thematic analysis on additional helpful aspects of the virtual breadboard.

The thematic analysis of student survey free response questions revealed several common themes regarding what students found helpful in their experience with the virtual breadboard. One prominent theme that emerged was the value of helpful error messages. Students appreciated clear and informative feedback provided by the system when they encountered errors or made mistakes while using the breadboard. This feature helped them identify and rectify their errors, enhancing their learning process. Another theme that emerged was the equitable nature of the virtual breadboard. Students expressed satisfaction with the platform's ability to provide an equal learning environment for all, irrespective of their physical resources or backgrounds. The virtual breadboard leveled the playing field and allowed students to engage with electronics and circuitry in an inclusive manner. Lastly, the overall design of the breadboard was another significant theme identified. Students commended the user-friendly interface, intuitive layout, and ease of navigation. The thoughtful design of the virtual breadboard contributed to a positive user experience, enabling students to focus on learning and experimentation.

Free Response Question	**Keywords and Phrases**	**Inferred Theme**
Suggestions for future improvements	Lock ability, make unable to edit breadboard when testing, provide ability to lock, save configurations, separate edit and interact modes	Ability to lock components
	Increase time limit, 3 minute time constraint, 3 minute timer	Increasing time limit
	Automatically submit designs, more intuitive submits, submit on switch toggling, clicking submit is tedious	Synchronized submit button features
	Further documentation, adding more supported components, analog circuits, saving and loading XML	Additional virtual breadboard features

Table 3.8: Thematic analysis on suggestions for improvements of the virtual breadboard.

Inferred themes from student suggestions on future improvements involve user interface refinements to continue to enhance the overall user experience of the virtual breadboard. Among these include a way to lock components, which as previously mentioned, provides a way to limit the number of unintentional components drags. Additional suggestions for improvements involve creating more usable ways to enhance breadboard features, including shareability features for saving and loading XML files of breadboard designs for future use, and additional support of components to include analog circuitry.

3.3.3 Overall Survey Results

The survey question responses and corresponding quantitative data for statistical analysis are given in the table below.

Question Category	Specific Question	Mean Response
Overall Virtual Breadboard Experience	The virtual breadboard is intuitive.	3.60
	The virtual breadboard is similar to a physical breadboard.	4.45
Virtual Breadboard Equitable Access	It is convenient to access this virtual breadboard at any location with an internet connection.	4.47
	It is convenient to access this virtual breadboard at any time that fits my schedule.	4.50
Individual Breadboard Features	Dragging components in this virtual breadboard is similar to that of placing components in a physical breadboard.	3.33
	Dragging components helped in my understanding of how breadboards work.	3.04
	The error messages displayed were helpful.	3.11
	The ability to draw wires was helpful in my experimentation.	3.36

	The ability to draw differently colored wires was helpful.	4.22
Virtual Breadboard GPIO	Interfacing the virtual breadboard's GPIO was similar to that of a physical breadboard.	3.75
	The virtual breadboard helped me understand the FPGA GPIO interface	3.68

Table 3.9: Student survey quantitative results.

For the question category for the *Overall Virtual Breadboard Experience*, students indicated that the virtual breadboard experience was intuitive and performed similarly to that of a physical breadboard that they had used in pre-requisite courses, with an overwhelming majority of students indicating responses at a 4 (Agree) or 5 (Strongly Agree) in the Likert Scale. Although students in the course assignment were not asked to create overly complex breadboard designs to aid in their SystemVerilog implementation, the quantitative result in this section helps provide insight into the overall usability aspect of a virtual breadboard, despite differences in the user experience elements needed through conversion of simple tasks into an interface on a computer screen. It provides a promising step that virtual breadboards controlled through a mouse and keyboard implementation is a viable interface for virtualizing key laboratory hardware.

The question category for *Individual Features* provides the most mixed results between all categories, though still siding favorably toward the positive responses of the virtual breadboard. Here, differences are also noticeable between the deployment in Academic Term 1 and Academic Term 2, with Academic Term 2 slightly favoring the breadboard more than Academic Term 1, many of which due to continued usability improvements presented to the breadboard during deployments. In particular, the most noticeable feature difference between Academic Term 1 and Academic Term 2 is the ability to asynchronously create the breadboard design without a timer constraint. The benefits of implementing it this way is twofold: one, it provides additional time for students to check their designs and use the individual breadboard features at their own desired pacing, and two, it allows students to customize their breadboard in parallel to writing their SystemVerilog implementation. Analysis for suggestions of improvements

additionally provides insight on achievable ways to better expand the usability aspect of the virtual breadboard to increase the quantitative response of this section in future iterations.

The question category for *Virtual Breadboard Equitable Access* provides the highest quantitative responses of all categories, thus highlighting a promising strength of virtualizing hardware. Students heavily support the time and location flexibility that the virtual breadboard offers, thus continuing to highlight the important nature of virtualizing and providing equitable access for remote laboratories. While students may have different preferences for where and when they like to complete course assignments, virtualizing the hardware allows additional flexibility on the place and time, something that traditional course offerings and physical hardware does not do well. Even in the post-pandemic era, students prefer the freedom of a blended learning experience.

The question category for *Virtual Breadboard GPIO* is among the most important in the study and provides the largest differentiating virtual breadboard feature from existing simulation based online breadboards. Here, the RHL-Butterfly interfaces with physical hardware that is live streamed back to the user to create an immersive, fully remote laboratory experience. An overwhelming majority of students responded positively in both Academic Term 1 and Academic Term 2 on the questions. This interface is what allows the virtual breadboard to become scalable for many different embedded systems courses; the breadboard can be used solely by itself, or it can be used to aid in GPIO manipulation for different microcontrollers and microprocessors. The virtual breadboard was able to communicate with the students' SystemVerilog designs appropriately, and responses from the FPGA boards were then captured and broadcasted back to the user. The high qualitative response for this section brings promising insight on the future of virtual breadboards to complement or replace existing traditional hardware in STEM curriculums.

Chapter 4. FUTURE WORK

As the virtual breadboard continues to evolve and be used in a variety of different digital logic design courses, student feedback and opinions of the breadboard will continue to be monitored, resulting in a larger amount of survey samples. The RHL-Butterfly is deployed and has been tested by students in two different offerings of the same engineering curriculum, which uses only the SPDT toggle switches and the virtual LEDs. Future offers that take advantage of the digital logic gate integrated circuits in introductory digital logic design courses are expected within the next year, resulting in larger awareness and higher consumer testing to collect samples of student surveys and feedback.

Additional rudimentary logic gate integrated circuits will continue to be added on the virtual breadboard, such as NOR, NAND, and XNOR logic gates. These gates were initially left out of the current iteration's design to promote student breadboard experimentation to chain NOT gates with the dual input logic gates. However, the available space on the virtual breadboard can be a limiting factor for some designs, and by adding a larger collection of supported circuit devices, it helps promote design space efficiency that allows for more complex breadboard designs in a smaller board estate.

The next iteration will also feature analog circuit devices, allowing users to take advantage of ADC and DAC peripherals on the target hardware. By doing so, we can expand on the supported integrated circuits for the virtual breadboard with simulated support for operational amplifiers, BJTs, and other passive circuit elements. In addition, we aim to provide support for virtual oscilloscopes and other virtual laboratory debugging tools to help students enhance their design creativity. This provides opportunities for students to explore all applications that embedded system design has to offer.

Chapter 5. CONCLUSION

As the educational world slowly reverts to the pre-pandemic practices, integrating remote laboratories and tools to the education curriculum brings greater opportunities for equitable access for STEM curricula; remote education is here to stay, even in a post-pandemic world. The RHL-Butterfly delivers one practical solution for embedded systems and engineering education. Using a virtual breadboard accessible through a breadboard graphical user interface via the internet, the simulated breadboard interfaces with real-world hardware to preserve the real-time metastability concepts that the physical hardware possesses.

The virtual breadboard captures the user created breadboard design by parsing through the 2D mapping of the breadboard state and representing the design as a 1D string. The 1D string is then transmitted through POST to a Raspberry Pi server to a Raspberry Pi Pico. The Raspberry Pi Pico then processes the digital computation of the electrical node and GPIO states. Depending on the constructed breadboard circuit, the Raspberry Pi Pico takes multiple update cycles to fully saturate the GPIO configuration. Once saturated, the GPIO information is then sent over to the target hardware, whose GPIO states are then captured via a livestream, displayed back over to the frontend client for the user. This creates an immersive remote laboratory experience and mimics the behavior of the virtual breadboard interfacing directly over to the target hardware.

Virtualizing a free, open-source breadboard experience for engineering education is fundamental in encouraging additional participation in STEM, paving the way for greater diversity, enhanced creativity, and larger innovations within our field. This virtualized breadboard thesis research to practice addresses current limitations of equitable engineering education access due to varying student situations, including financial costs, seasonal illnesses, and disabilities, delivering an approach to promote STEM participation to the under-privileged. By bridging the gap between education and technology, RHL-Butterfly has become an integral part of the remote laboratory experience within the Remote Hub Lab, and fundamental toward the integration to both college and pre-college STEM courses of Digital Logic Design curriculum through the Remote Hub Lab's RHL-BEADLE project [22].

Chapter 6. REFERENCES

[1] Rabab Ali Abumalloh, Shahla Asadi, Mehrbakhsh Nilashi, Behrouz Minaei-Bidgoli, Fatima Khan Nayer, Sarminah Samad, Saidatulakmal Mohd, Othman Ibrahim, "The impact of coronavirus pandemic (COVID-19) on education: The role of virtual and remote laboratories in education," Technology in Society, vol. 67, 2021, https://doi.org/10.1016/j.techsoc.2021.101728.

[2] A. K. Mohammed, H. M. El Zoghby and M. M. Elmesalawy, "Remote Controlled Laboratory Experiments for Engineering Education in the Post-COVID-19 Era: Concept and Example," *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, Giza, Egypt, 2020, pp. 629-634, doi: 10.1109/NILES50944.2020.9257888.

[3] R. Heradio, L. de la Torre, D. Galan, F. J. Cabrerizo, E. Herrera-Viedma, and S. Dormido, "*Virtual and remote labs in education: a bibliometric analysis*," Computers & Education, vol. 98, pp. 14–38, Jul. 2016, doi: 10.1016/j.compedu.2016.03.010.

[4] Ogot, M., Elliott, G. and Glumac, N. (2003), An Assessment of In-Person and Remotely Operated Laboratories. Journal of Engineering Education, 92: 57-64. https://doi.org/10.1002/j.2168-9830.2003.tb00738.x

[5] W. -S. Soh, "Experiential Learning Through Remote Electrical Engineering Labs During the COVID-19 Pandemic," *2021 IEEE International Conference on Engineering, Technology & Education (TALE)*, Wuhan, Hubei Province, China, 2021, pp. 01-05, doi: 10.1109/TALE52509.2021.9678756.

[6] Hussein, Rania, and Wilson, Denise, "*Remote versus In-hand hardware laboratory in digital circuits courses",* American Society for Engineering Education ASEE conference, Electrical and Computer Engineering Division, July 26-29, 2021.

[7] F. Atienza and R. Hussein, "Student Perspectives on Remote Hardware Labs and Equitable Access in a Post-Pandemic Era," *2022 IEEE Frontiers in Education Conference (FIE)*, Uppsala, Sweden, 2022, pp. 1-8, doi: 10.1109/FIE56618.2022.9962440.

[8] Zhao, Y., Watterston, J. The changes we need: Education post COVID-19. *J Educ Change***22**, 3–12 (2021). https://doi.org/10.1007/s10833-021-09417-3

[9] Zheng, M., Bender, D. & Lyon, C. Online learning during COVID-19 produced equivalent or better student course performance as compared with pre-pandemic: empirical evidence from a school-wide comparative study. *BMC Med Educ* **21**, 495 (2021). https://doi.org/10.1186/s12909-021-02909-z

[10] R. Hussein, B. Chap, M. Inonan, M. Guo, F. L. Monroy, R. C. Maloney, S. A. Ferreria, and S. J. Kalisi, “Remote Hub Lab – RHL: Broadly Accessible Technologies for Education and Telehealth,” *2023 International Conference on Remote Engineering and Virtual Instrumentation*.

[11] Hodges, C., Moore, S., Lockee B., Trust, T., & Bond, A. “The Difference Between Emergency Remote Teaching and Online Learning,” 2020. Available: https://er.educause.edu/articles/2020/3/the-difference-between-emergency-remote-teaching-and-online-learning

[12] N. Glass, “*Java Digital Breadboard Simulator: A simulator for educational electronics environment,*” 2002. Available: https://muchlas.ee.uad.ac.id/downloads/nicholas%20glass-jbreadboard.pdf

[13] Autodesk. Tinkercad. [Online]. Available: https://www.tinkercad.com/

[14] P. La Rocca, F. Riggi, and C. Pinto, “*Remote teaching Arduino by means of an online simulator,*” Physics Education, 2020. doi: 10.1088/1361-6552/abaa21

[15] P. Orduña, L. Rodriguez-Gil, J. Garcia-Zubia, I. Angulo, U. Hernandez and E. Azcuenaga, "LabsLand: A sharing economy platform to promote educational remote laboratories maintainability, sustainability and adoption," *2016 IEEE Frontiers in Education Conference (FIE)*, Erie, PA, USA, 2016, pp. 1-6, doi: 10.1109/FIE.2016.7757579.

[16] S. Li, H. Wang, L. Rodriguez-Gil, P. Orduña, and R. Hussein, “*FPGA Meets Breadboard: Integrating a Virtual Breadboard with Real FPGA Boards for Remote Access in Digital Design Courses.*” in *Online Engineering and Society 4.0*, 2020, pp. 144-151. https://doi.org/10.1007/978-3-030-82529-4_15

[17] D. May, B. Reeves, M. Trudgen and A. Alweshah, "The remote laboratory VISIR - Introducing online laboratory equipment in electrical engineering classes," *2020 IEEE Frontiers in Education Conference (FIE)*, Uppsala, Sweden, 2020, pp. 1-9, doi: 10.1109/FIE44824.2020.9274121.

[18] L. Rodríguez-Gil, J. García-Zubia, P. Orduña and D. Lopez-de-Ipiña, "*An Open and Scalable Web-Based Interactive Live-Streaming architecture: The WILSP Platform,*" in *IEEE Access*, vol. 5, pp. 9842-9856, 2017, doi: 10.1109/ACCESS.2017.2710328.

[19] M. Tawfik *et al*., "Virtual Instrument Systems in Reality (VISIR) for Remote Wiring and Measurement of Electronic Circuits on Breadboard," in *IEEE Transactions on Learning Technologies*, vol. 6, no. 1, pp. 60-72, Jan.-March 2013, doi: 10.1109/TLT.2012.20.

[20] “Raspberry Pi Documentation,” 2023. Available: https://www.raspberrypi.com/documentation/pico-sdk/

[21] Ada, L., “Headers,” 2016. Available: https://learn.adafruit.com/ladyadas-learn-arduino-lesson-number-0/headers

[22] R. Hussein, R. Maloney, P. Orduna, J. Ander Beroz, L. Rodriguez-Gil, “RHL-BEADLE: Bringing Equitable Access to Digital Logic Design in Engineering Education”, American Society for Engineering Education ASEE conference, 2023.