regvue
Improving Hardware Register Documentation Tooling

Josh Geden; Rob Donnelly

2022-08-15

Table of Contents

1. Abstract

Hardware documentation at JPL has historically been generated by hand and accessed through shared PDF or Word documents. This can lead to issues where the documentation does not keep pace with the actual hardware implementation. It also leads to documents that are difficult to use and lack the necessary features to accelerate the hardware debugging process. To address these issues, a hardware documentation tool called *regvue* was created. This tool is available as a web application and is capable of building advanced documentation sites from data files that implement the Register Description Format. Documentation can now be automatically updated by making changes to the data files, ensuring that engineers always have access to current information in an easy-to-read format. Features such as full-text search and automatic bit encoding/decoding are also provided. Future features will include enabling *regvue* to interface with hardware and directly read and decode register values. By providing higher quality documentation and advanced features with *regvue*, hardware engineers will spend significantly less time during the debugging process.

2. Introduction

When engineers at JPL need to consult Field Programmable Gate Array (FPGA) hardware documentation, they traditionally have turned to PDF or Word documents created and maintained by FPGA design engineers that are hosted on internal file-sharing websites. From these websites, FPGA hardware test engineers can either preview the document online or they can download the file and open it locally. Because online file viewers lack certain features and are typically slower to use, most engineers download these documentation files and use them locally.

This creates an issue when the design engineer uploads a new version of the documentation but test engineers continue to use their locally-downloaded documentation file, which has since become outdated. This mismatch can be difficult to notice and can incur significant delays, and therefore additional costs, during the FPGA testing and debugging process.

In addition to this, the documentation provided by these documents is static and does not support the actual testing and debugging workflow of hardware engineers. For example, say there is a field that takes up the bottom (least-significant) 6 bits of a 32 bit register. This field represents the number of errors a certain device encounters. A typical workflow to obtain the number of errors may therefore look like:

1. Read out the value of the register using external tools as a big-endian hexadecimal value
2. Convert the big-endian value to a little-endian value
3. Convert that value from hexadecimal to binary
4. Isolate the bottom 6 bits of that binary value
5. Convert that value to decimal

This manual manipulation and conversion of register and field values is incredibly error-prone and unnecessarily increases the complexity of the already complex hardware testing and debugging process.

These issues with JPL's existing hardware documentation strategy is what inspired the development of *regvue*. Available as both a web and desktop application, *regvue* utilizes a design-agnostic JSON schema called the Register Description Format (RDF) to generate easy-to-use and modern hardware documentation.

Design engineers can use existing register automation tools to generate RDF files from existing documentation specifications. These files can then parsed by *regvue* to produce a website with all the features of classic online technical documentation sites as well as additional features specific to the FPGA testing and debugging workflow.

FPGA hardware test engineers no longer have to worry about accessing out-of-date documentation, because FPGA design engineers can host their RDF files through online repositories that *regvue* can access, so any update to the design can be automatically reflected in the documentation site accessed by test engineers. Test engineers also no longer have to manually parse register values, because *regvue* includes an interactive bit table for every register that can automatically decode register values into meaningful representations and can automatically convert values between hexadecimal, decimal, and binary representations. The documentation provided by *regvue* will therefore significantly reduces the costs and delays associated with testing and debugging FPGA hardware.

3. Methods

The following sections describe in a high-level manner the different aspects of the *regvue* application, the processes used to create it, and the surrounding technologies necessary to run it. It will touch on the different problems encountered and how these problems were resolved.

3.1. Development

regvue was developed by a small team of 2 developers. New feature requests, potential bugs, and any discussion on design implementation was managed using GitHub Issues. Any changes to the codebase were tracked and implemented incrementally using GitHub Pull Requests.

regvue adheres to semantic versioning to track the addition of new features and to introduce backwards-incompatible features or changes. New features are also tracked in a changelog file.

When *regvue* reached a point where a new release was desired, first a GitHub Tag was created to mark the commit in the repository that the release corresponds to. Then a compressed production bundle of the app would be published as a GitHub Release. This Release would come with a summary of all the new changes recorded in the changelog file as release notes.

After creating a new release, the official *regvue* deployment maintained by the *regvue* development team is updated to this new version.

3.2. Schema

regvue expands upon previous work that resulted in the creation of the Register Description Format (RDF) schema. The RDF provides a design-agnostic JSON-encoded hardware design formalization that can be used to document FPGA hardware systems.

On different projects and missions, FPGA design engineers make use of different file formats and file types, such as Word DOCX files or Tcl files, to document their designs. The RDF was created to bridge the gap between these different formats and file types and to provide a general-use data format that can be used by external applications. Different register automation solutions were then created that are capable of generating RDF files from the different types of hardware documentation specifications.

Because the RDF is designed to be used by external applications, it strictly adheres to semantic versioning to ensure version compatibility.

All RDF files begin as JSON-encoded files with three top-level objects.

The `schema` object defines the name and version of the schema being used. Only v1.x of the `register-description-format` is currently supported by *regvue*, but this meta data was included to allow for future data formats to be supported in the future.

Example `schema` object

```
"schema": {
    "version": "v1",
    "name": "register-description-format"
}
```

The `root` object provides information about the overall FPGA design being represented. This includes the name, version, and descriptive doc text. It also includes a list of ids for all root-level elements (or root "children").

Example `root` object

```
"root": {
    "version": "v1.0",
    "desc": "Example Design",
    "doc": "This is an example design.",
    "children": [ ... ],
}
```

The `elements` object is a map of all element within the system.

Example `elements` object with a single `reg` type element

```
"elements": {
    "registerA": {
        "id": "registerA",
        "name": "registerA",
        "type": "reg",
        "offset": "0x0",
        "doc": "Register A - an example register",
        "fields": [
            {
                "name": "example_field",
                "access": "ro",
                "lsb": 0,
                "nbits": 32,
                "doc": "Example field"
            }
        ]
    }
}
```

The RDF uses a dot notation to manage element hierarchy. If an element has the name `register` and its parent element is named `block` which has no parent element itself, then the `register` element has the id `block.register`.

In order to ensure any RDF files opened in *regvue* conform to this schema, files are validated using JSON Schema when they are first loaded. JSON Schema is a standardization that can be used to define what a JSON document must look like, ways to extract information from it, and how to interact with it. Different validation libraries exist that support the JSON Schema standard and can be used to automatically validate that JSON files adhere to a specific schema.

3.3. Application

When developing *regvue*, the two highest priority goals were interactivity and portability.

There were already existing solutions in the form of Word documents and static auto-generated HTML pages, but these solutions lack interactivity.

Word document specifications also lack portability between different operating systems due to formatting quirks with Microsoft Office and have issues regarding version control. Test engineers will typically have a local copy of Word document specifications that they use for reference. In the past, design engineers have made changes to FPGA designs and updated their documentation to reflect that, but hardware test engineers continued to use their out-of-date local documentation, which incurred unnecessary delays in the hardware testing and debugging process.

We considered creating a desktop app with Python and the TK GUI library. This would have been a portable solution, but more difficultly so, because there would be overhead in terms of users having to install Python to run the application. It also makes updating *regvue* much more complex because users would be running local executables and would therefore suffer from the same version control issue that affects local documentation.

Based on the shortcomings of these implementations, we decided to create *regvue* as a web application. This allows us to include interactivity by using JavaScript within the app and it is incredibly portable because users can access it from any browser on any type of OS.

A web application also has the benefit of having a lower-barrier to entry. There is no installation necessary to use *regvue*, so users that do not interact with the hardware testbed, such as design & verification teams and software teams, are more likely to use the application.

3.3.1. Frontend

In this context, the frontend of the *regvue* app refers to everything that the user can see and interact with, and the code necessary to support that interactivity. In order to build the frontend user interface (UI) of *regvue*, we used the Vue framework with Typescript.

Using a framework like Vue simplifies the process of developing a web application. It provides a declarative model, meaning when the state of the website changes, such as when a user clicks on a button or inputs a value in a text box, the UI automatically updates to match the new state. It also provides a component model, meaning sections of code can be encapsulated in modular components that can be reused multiple times throughout the application.

Similar frontend frameworks, such as Angular or React, also could have been used to create *regvue*. We made the decision to use Vue because of its use of native HTML templates to build UIs (as compared to React's use of JSX) and because Vue follows a progressive development model (as compared to Angular's more opinionated MVC-based design).

We also used Typescript instead of plain JavaScript to improve the ease of development and maintainability of the code base. Typescript allows the project to have well defined type interfaces that improve code readability and will make returning to the source code easier in the future.

regvue uses Tailwind CSS, a CSS utility framework that provides composable CSS classes to functionally build modern styles. Originally, *regvue* was built using pre-stylized components from the PrimeVue component library. PrimeVue provides pre-made components that can be used to quickly build a web app, but at the cost of not being able to modify the styling of those components very easily. Tailwind is also incredibly performant and will automatically remove unused CSS classes to ship the smallest possible CSS file, meaning it has a much lighter footprint than PrimeVue.

3.3.2. Bit Manipulation

One of the most pressing issues that *regvue* was designed to resolve was to remove the need for hardware test engineers to manually manipulate register and field values.

In one instance, manual hardware testing found some unexpected behavior in which status bits would unexpected clear or not clear. Eventually it was found that the issues were caused by bad input. In one case, the test engineer wrote the value 0x0001000 when they should have written 0x00010000 (one more 0 digit at the end of the value). In another case, the engineer wrote 00010030 instead of 0x00010030. Here, the lack of a 0x prefix caused the value to be interpreted as octal. In both cases, the perceived differences are subtle (and difficult to catch), but the effective differences are huge. These issues required several days to resolve and could have been easily avoided if hardware engineers did not have to manually manipulate register values or if they had access to a tool that could automatically decode register values.

To prevent this issue in the future, every register element shown in *regvue* comes with an interactive bit table that automatically decodes the register value into its different fields.

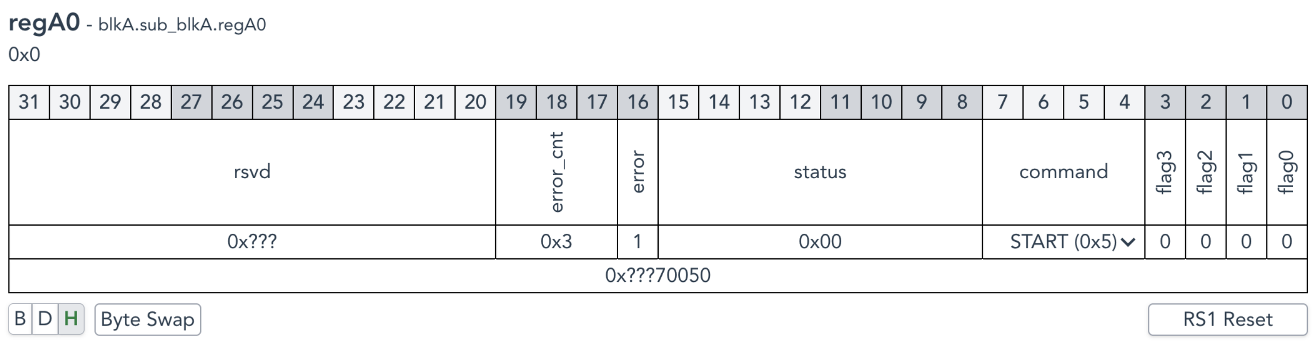
Using this table, users can enter new register or field values. If the user enters a new register value, then the fields are automatically updated based on the new value. And if the user enters a new value for a specific field, the register value will be updated based on that new value. These inputs are also automatically verified to ensure maximum size is not exceeded and that no invalid characters are included.

The table includes a set of buttons that can be used to swap between binary, decimal, and hexadecimal representations.

It also includes a toggle button that will byte swap the register value. This allows for big-endian values to be automatically converted to little-endian, and vice versa, before being broken decoding into field values.

Registers can also come with different reset values, that are triggered by certain events, such when the device is powered on or when a hot reset is triggered. The bit table provides a button and dropdown menu that can reset field values to any associated reset state.

Certain field values may correspond to named symbolic states that can give more meaning and context to raw encoded values. For example, if an error field has a value of "0x1" that might correspond to an "Error" state. The bit table supports enumerated field values and can automatically map these specific numeric values to named symbolic states.



regA0 - blkA.sub_blkA.regA0

0x0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rsvd												error_cnt			error	status								command				flag3	flag2	flag1	flag0
0x???												0x3			1	0x00								START (0x5)				0	0	0	0
0x???70050																															

Figure 1. Bit table for an example register

3.3.3. Search

FPGA designs can have over hundreds of hardware elements to document, so including extensive search functionality that can be used to quickly find elements was one of the first features to be added to *regvue*.

An important restriction on how we could implement search functionality was the need for export-controlled information to remain secure (see Restricted Information for more information). This means we could not use popular search providers such as Elastic Search or Algolia, because they require transmitting the data to an external server. These providers also usually charge for their services, and *regvue* is committed to be accessible as free and open-source software.

For these reasons, we decided to implement a client-side search using the Lunr search library. Lunr provides simple and extensible search functionality that has no external dependencies and can run completely within the browser, meaning no export-controlled information will ever be transmitted to an external service.

After users load a RDF file, a Lunr search index object is created that can be used to search for any hardware element by id, name, offset, or description text. When the user provides a search query by typing in the search box, the index object will return a list of element ids that most closely correspond to the search text.

There are some drawbacks to using a client-side search. There is additional load time necessary to build the search index on app load and it can take much more time to search the index with a given search query compared to external search providers. However, because the data being searched is just plain text, client-side search has scaled relatively well. For the Europa Compute Element design, which is a composition of 6 FPGA designs, the time to build the search index and to search the entire index typically takes less than 1 second.

3.3.4. Routing

regvue is implemented as a single-page application, and therefore needs to use a router to control which pages are displayed when users navigate to different URLs. In traditional dynamic web applications, when the user navigates to a specific URL, the server will respond with a specific HTML file. But with a single-page application, an object called the router will instead conditionally render different components based on the URL.

As a progressive framework, Vue allows developers to opt-in to different levels of complexity, and therefore does not provide a router out of the box. However, the official Vue Router library is incredibly simple to add to an existing project since it follows a plugin-style architecture.

When the app first loads, a router object is created that comes with a predefined set of routes to handle. *regvue* currently has three distinct page views that the router can display based on the URL.

The first page view that most users of *regvue* will see is the open page. This page corresponds to the `/open` URL and provides users with input boxes to load a RDF file from the local filesystem or from a URL.

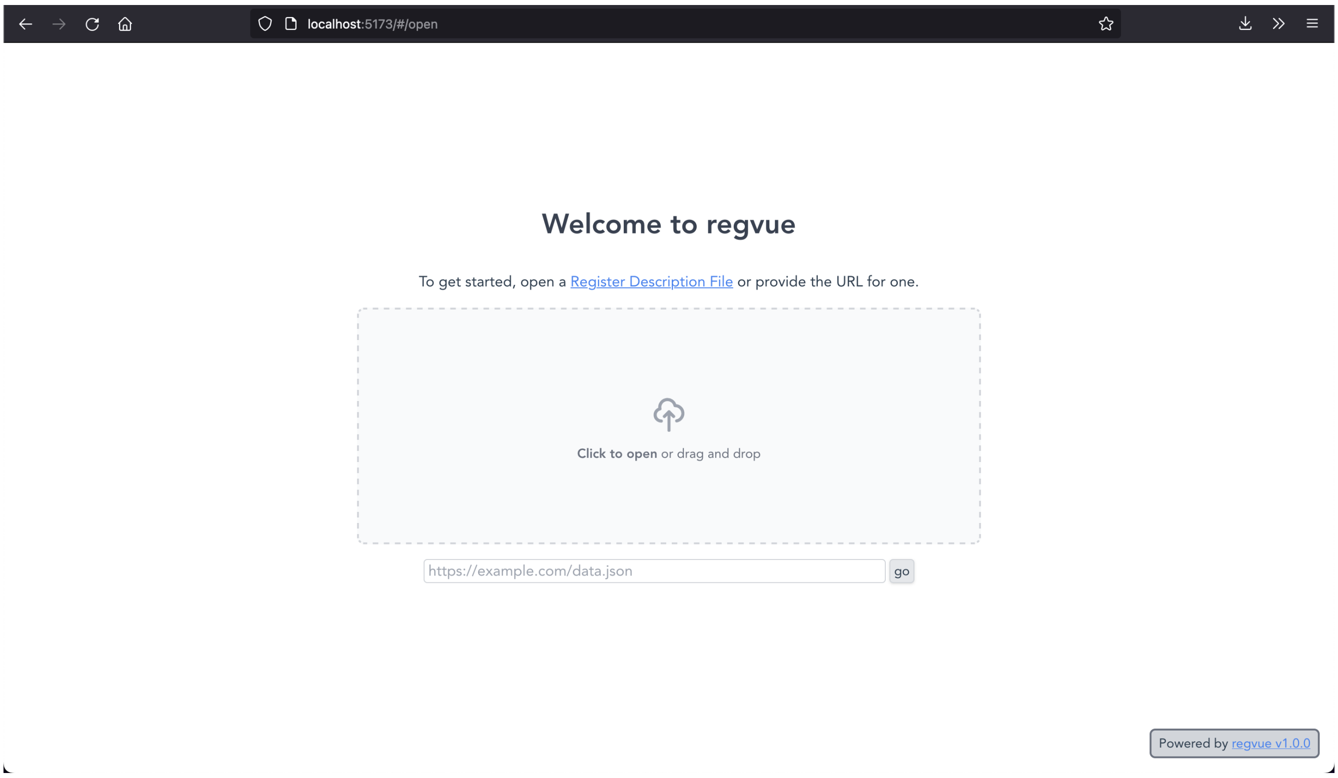
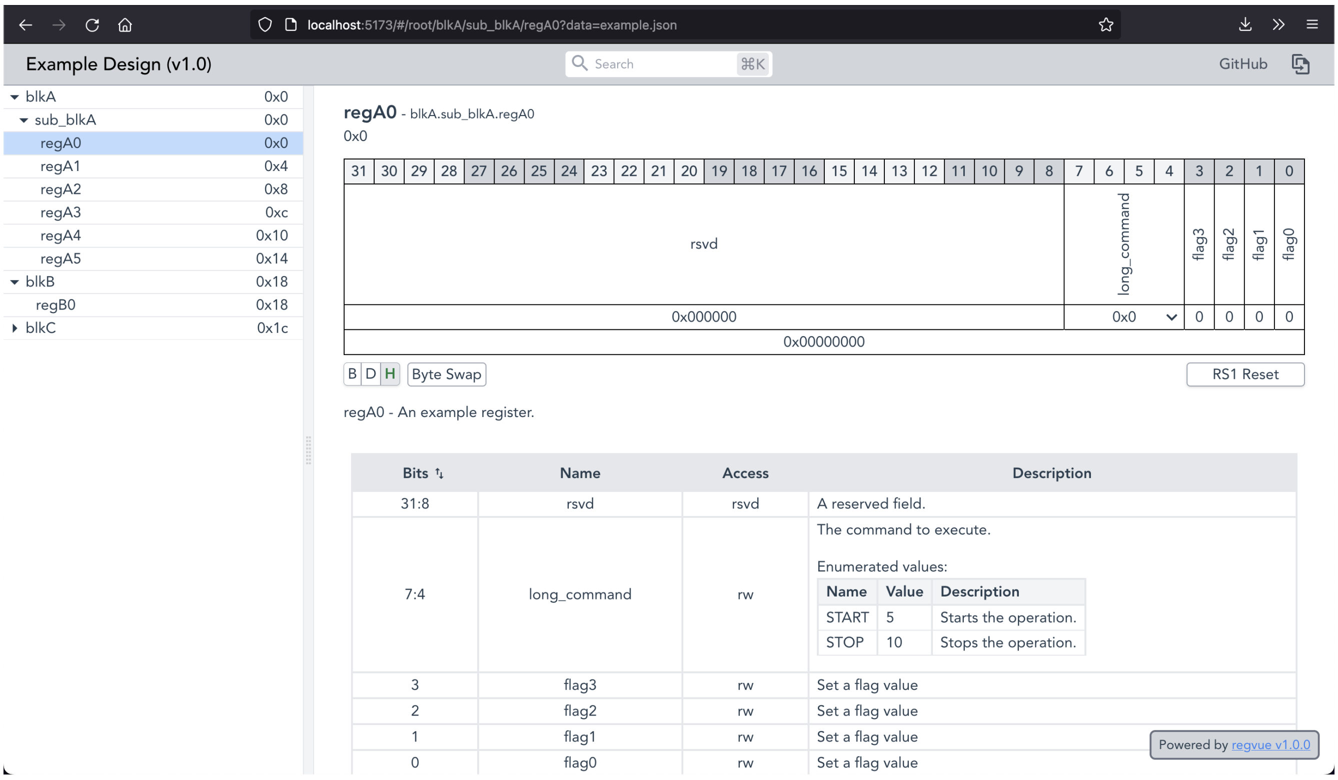

localhost:5173/#/open
Welcome to regvue
To get started, open a Register Description File or provide the URL for one.
Click to open or drag and drop
https://example.com/data.json
go
Powered by regvue v1.0.0

Figure 2. Open page

Once the user opens a RDF file, the element page is shown. This is the main view of *regvue* and displays the documentation information about the different design elements. It also includes the navigation menu and the header.

URLs for the element view start with `/root`. Then to view a specific element, the element id is given in the form of a URL. So to view an element with id `system.board.ctrl`, the URL would be `/root/system/board/ctrl`.



localhost:5173/#/root/blkA/sub_blkA/regA0?data=example.json

Example Design (v1.0) Search ⌘K GitHub

blkA	0x0
sub_blkA	0x0
regA0	0x0
regA1	0x4
regA2	0x8
regA3	0xc
regA4	0x10
regA5	0x14
blkB	0x18
regB0	0x18
blkC	0x1c

regA0 - blkA.sub_blkA.regA0
0x0

31:8	7:4	3	2	1	0
rsvd	long_command	flag3	flag2	flag1	flag0
0x000000	0x0	0	0	0	0

0x00000000

B D H Byte Swap RS1 Reset

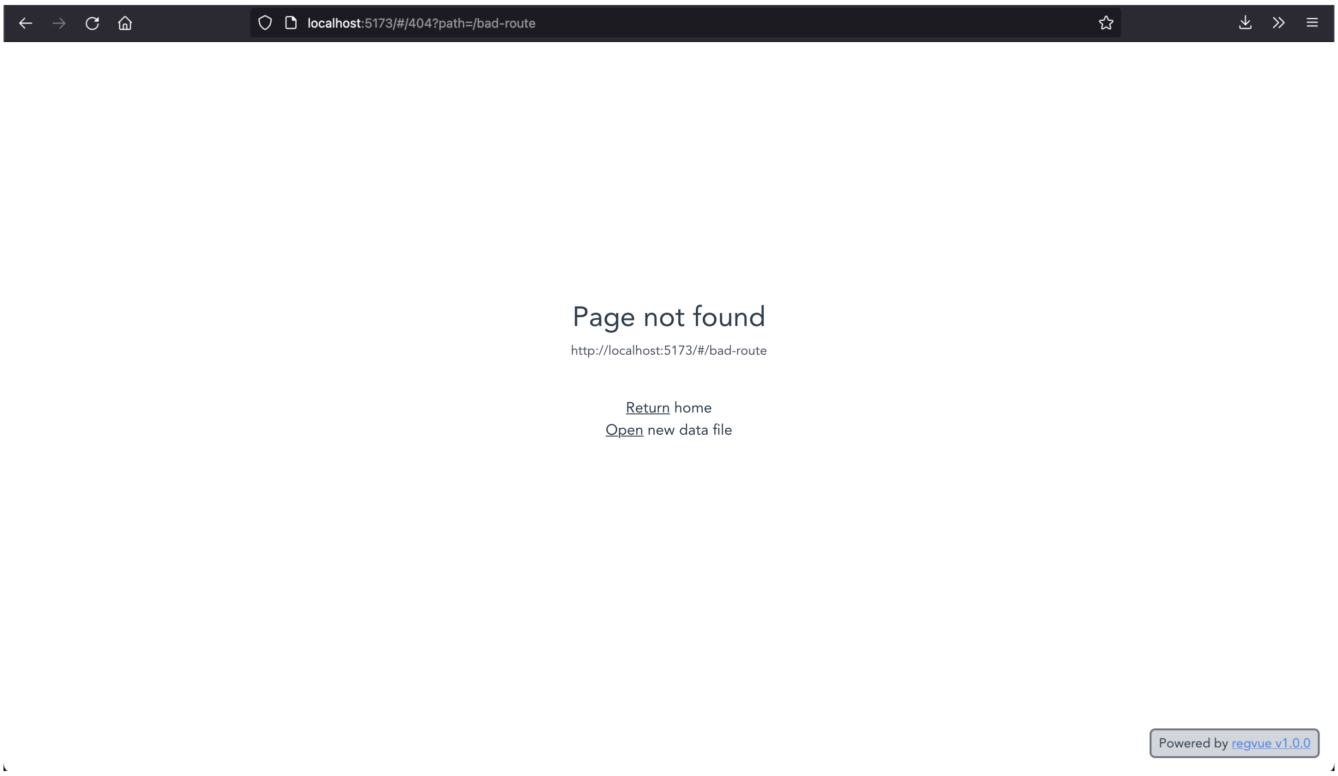
regA0 - An example register.

Bits	Name	Access	Description
31:8	rsvd	rsvd	A reserved field.
7:4	long_command	rw	The command to execute. Enumerated values: Name / Value / Description START / 5 / Starts the operation. STOP / 10 / Stops the operation.
3	flag3	rw	Set a flag value
2	flag2	rw	Set a flag value
1	flag1	rw	Set a flag value
0	flag0	rw	Set a flag value

Powered by regvue v1.0.0

Figure 3. Element page with an example register

The final page view is the 404 page. This view is displayed when a user either enters a URL that does not correspond to a pre-defined route or tries to navigate to an element that does not exist.



localhost:5173/#/404?path=/bad-route

Page not found

http://localhost:5173/#/bad-route

Return home
Open new data file

Powered by regvue v1.0.0

Figure 4. 404 page

3.3.5. Data Store

regvue uses Pinia, a store library specifically designed for use with Vue, to store and maintain all cross-component data that is used in the app. Normally, data within Vue apps must be directly passed from parent components to child components, but having data that is accessible throughout the entire app can help simplify this data hierarchy.

When *regvue* first loads, a Pinia store object is created. When the user then opens an RDF file, the store object parses the raw JSON and generates a map that stores all hardware elements (i.e. registers, blocks, or memory segments) within the FPGA design as formatted TypeScript objects.

This greatly simplifies accessing information about different elements from the different components within the file. Instead of passing information about all the different elements from every parent component to every child component, *regvue* can instead obtain the current element's id from the URL, and then can use that id to access information about the element from the store object.

3.3.6. Include Elements

It is possible for FPGA designs to consist of multiple FPGA sub-designs that are grouped together due to related function or location. An issue we encountered during development was the need to continually combine FPGA sub-designs for larger combined designs. For example, the Europa Input/Output (EIO) and the Europa Memory (EMEM) FPGAs had both been specified using the RDF. But the overall Europa Compute Element (ECE) contains both of these sub-designs. Originally an external tool was used to combine the two sub-design RDF files into a single combined file. This was not desireable because it meant an extra tool would need to be maintained and that there would duplicated information being stored.

To address this issue, we introduced the concept of an Include Element. Rather than specifying an element as a register, block, or memory section, hardware design engineers can instead specify an element of type `include` and then provide a URL. When *regvue* loads the RDF file, it will attempt to fetch an RDF file from the given URL and insert it into the parent RDF file.

Example `include` element

```
"ece.eio": {
    "name": "eio",
    "id": "ece.eio",
    "type": "include",
    "offset": "0xc0e00000",
    "url": "eio.json" ①
}
```

① This is a relative URL property. If the parent json file is at `https://example.com/ece.json`, then this url corresponds to `https://example.com/eio.json`

This drastically simplifies the process of building combined designs and removes the need for an external tool to combine RDF files.

3.3.7. Testing

Testing is incredibly important for ensuring previously implemented features continue to work. Unit tests are typically used to test individual functions in isolation. End-to-end tests or integration tests are used to test how an application functions when its different modules are integrated and working together.

regvue uses the Vitest unit testing framework to test utility functions. These tests provide an input and expected output to different isolated functions. The testing framework will then call the given functions with the provided inputs and ensure it matches the provided outputs. For example, there is a utility function that is used throughout the *regvue* app to turn a string representation of a value into a number. So "0x2", "0b10", and "2" would all return the numeric value 2. There is a unit test that provides these three different inputs and asserts that the function returns 2 for all of them.

regvue uses the Cypress end-to-end testing framework to perform browser-based integration testing. These tests are more focused on testing the interactivity of the app, and often test multiple components at the same time. These tests are written by accessing HTML DOM elements, typically by the elements' ids, and then performing actions on those elements, such as clicking or typing, by calling Cypress functions.

An example of an integration test that *regvue* has includes typing a search term in the search box, selecting the first search result, and ensuring that the app then displays information about that search result. Other Cypress tests written for *regvue* include testing menu navigation, bit table value manipulation, and opening new RDF files.

3.4. Deployment

regvue has minimal hosting requirements and can be deployed as a static site on almost any hosting platform, including free services such as GitHub Pages and GitLab Pages. The *regvue* development team maintains a set of deployments using GitHub Pages that correspond to all major versions of *regvue*, meaning end users do not need to install or deploy *regvue* if they do not wish to.

3.4.1. Restricted Information

Most information regarding FPGA designs at JPL is subject to U.S. Export Regulations. This means that the information cannot be shared with foreign persons without prior approval. There are also additional sensitivity levels that require further restriction, such as Controlled Unclassified (CUI), Sensitive but Unclassified (SBU), or For Official Use Only (FOUO).

To allow for information that falls under these sensitivity levels to be usable with *regvue*, significant thought had to be given to ensure that users can only access information that they are authorized to. The *regvue* application itself does not include any sensitive information within its deployed build code, so the problem that we had to solve was to ensure any RDF files that users want to link to are limited to those who are allowed to access them, but still accessible by the *regvue* app.

Our use of GitHub Pages actually solved this issue for us with minimal overhead. We host *regvue* on JPL's GitHub Enterprise server using GitHub Pages, which is only accessible to U.S. persons and requires users to be logged in with a JPL GitHub account. When users then want to access a RDF file by URL, if that file is also hosted in a public repository on JPL's GitHub Enterprise server, no additional authentication is necessary because they are already logged in with their JPL GitHub account.

Information that falls under stricter sensitivity levels must be restricted further. To accomplish this, RDF files can be hosted in a private repository on JPL's GitHub Enterprise Server and then access can be granted to select individuals. When users attempt to load a file from these private repositories, their JPL GitHub account will be checked to ensure they are allowed access.

This use of GitHub Enterprise therefore means the official *regvue* deploy has built-in authentication with no additional code necessary.

3.4.2. Desktop Application

While *regvue* was primarily developed to be deployed as a web app, we have also been able to create executable binaries that are capable of running directly on Windows, Mac, and Linux operating systems as a desktop application. To accomplish this, we used the Tauri framework, which provides a cross-platform WebView rendering library that is capable of displaying a web-based frontend. While not officially supported yet, we plan to use these local executables in the future to add specific features to *regvue* that would be otherwise impossible due to browser limitations.

4. Results

Engineers working on the Europa Clipper and Mars Sample Return (MSR) missions have already integrated *regvue* into their workflows, and so far the tool has proved to be a great help.

> I have used regvue a lot during integration testing for Europa Clipper. There are integration tasks [where I have] to poke and peek at registers and the regvue tool allows me to quickly look up a register and test out different register values. Figuring out those register values can be quite challenging since it is broken down to 32 bits, but the regvue tool helps make that translation easier. It cuts down the time it would take to go through the document, put down on paper what the register should be and double checking the value. Also, with the tool being able to convert from binary to decimal to hexadecimal, it makes translating engineering values way easier. I hope to continue to use this on future projects such as MSR.
>
> — Brian Nguyen, Senior Electrical Engineer, Europa Clipper (348E)

> Regvue is the interactive register viewer I have dreamed about for years. It's a powerful tool to assist hardware designers, software designers, and end-users. I plan to use it on all of my flight FPGA designs going forward.
>
> — Ryan Stern, MSR SRL Motor Control Card FPGA Task Lead (349C)

Users have consistently remarked on how *regvue* drastically simplifies the hardware testing and debugging process.

The interactive bit table is perhaps the most popular feature amongst users and many have said they can't imagine ever having to return to manually manipulating register values.

regvue is not done being developed either. Futures plans include adding the ability to compare and provide diff highlighting for multiple potential values of a single register and to allow for real-time testbed integration.

5. Conclusion

JPL now has a robust hardware documentation solution that can be used to document any future hardware designs. The documentation provided by *regvue* is easy to access, easy to update, and provides advanced features that are capable of accelerating the hardware testing and debugging process. Hardware test engineers will no longer accidentally access out-of-date documentation or make errors trying to manually manipulate register and field values.

Design and hardware engineers who have used *regvue* have already demonstrated enthusiasm to continue to use the tool on future projects and hardware leads on both the Europa Clipper and MSR missions are eager to introduce the tool to more teams throughout JPL.

6. Acknowledgments

Special thanks to Rob Donnelly for his mentorship and guidance.
regvue was his vision, and it has been an extremely rewarding experience to help bring that vision to reality.

Thanks to Mike Thielman, Europa Clipper Avionics Systems Engineering Lead, and the Europa Clipper Mission for providing the funding and administrative support necessary to transform *regvue* from a proposal and prototype to a full-fledged hardware documentation solution.

Additional thanks to Ryan Stern and the hardware engineers in the Flight Software and Avionics Systems Group for their early adoption of *regvue* and for supporting its development by suggesting new features and identifying bugs.

And lastly, thanks to the Caltech Student-Faculty Programs Office and JPL for hosting the JPL Summer Internship Program and to all those who helped support this program.