Technische
Hochschule
Augsburg

Masterthesis

Field of Study
Computer Science

Development of a Schematic Viewer for Digital Designs

as an Open-Source Project

in the field of Efficient Embedded Systems

Author: Lukas Bauer

Examiner: Professor Dr. Gundolf Kiefer
Secondary Examiner: Professor Dr. Hubert Högl

Author:
Lukas Bauer
lukas.nassenhausen@googlemail.com

Technische Hochschule
Augsburg
An der Hochschule 1
86161 Augsburg
phone: +49 (0)821-5586-0
fax: +49 (0)821-5586-3222
info@tha.de

© 2025 Lukas Bauer

This work withe the title

»Development of a Schematic Viewer for Digital Designs - as an Open-Source Project«

written by Lukas Bauer is licensed under

Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA).
https://creativecommons.org/licenses/by-sa/4.0/deed.en

All results described in this work and available on the enclosed data carrier in the form of source codes, software, and concept drafts are licensed under the GNU General Public License Version 3.
http://www.gnu.de/documents/gpl.en.html

The LaTeX template is based on
http://f.macke.it/MasterarbeitZIP.

Contents

Table of Contents III

List of Abbreviations VII

List of Figures IX

List of Tables XI

Listings XII

1 Introduction 1

- 1.1 Motivation 1
- 1.2 Purpose of the Work 2
- 1.3 Outline of the Thesis 3

2 Fundamentals 3

- 2.1 File Formats for Circuit Representation 4
  - 2.1.1 The Electronic Design Interchange Format (EDIF) 4
  - 2.1.2 The VHSIC Hardware Description Language (VHDL) Format 4
  - 2.1.3 The Yosys Open Synthesis Suite (Yosys) Netlist Format . . . 5
- 2.2 Open Source Synthesis Toolchain 5
- 2.3 The WebAssembly Standard 5
  - 2.3.1 The WebAssembly Compiler emscripten 6
- 2.4 Graphical User Interfaces 6
  - 2.4.1 Qt Graphics Framework 6
  - 2.4.2 WebAssembly and Qt 7
- 2.5 Development Environment 7
  - 2.5.1 Continuous Integration 8
    - 2.5.1.1 Jenkins Automation Server 8
  - 2.5.2 Docker Container 8
  - 2.5.3 Testing Environment for Qt 8
  - 2.5.4 Distribution with AppImage 9

3 State of the Art **9**

3.1 Routing Libraries . . . 9

3.1.1 `elk/elkjs` . . . 10

3.1.2 `Open Graph Drawing Framework` . . . 11

3.1.3 Adaptagrams Libraries . . . 12

3.1.3.1 `libvpsc` . . . 13

3.1.3.2 `libcola` . . . 13

3.1.3.3 `libavoid` . . . 14

3.1.4 Conclusion . . . 14

3.2 Existing Schematic Viewers . . . 15

3.2.1 `yosys show` . . . 16

3.2.2 netlistsvg . . . 17

3.2.3 TerosHDL . . . 19

3.2.4 digitaljs . . . 20

3.2.5 Conclusion . . . 21

4 Overview **21**

4.1 Structure . . . 22

4.2 Libraries . . . 23

4.3 File formats . . . 23

4.3.1 Symbols . . . 23

4.3.1.1 Differences to netlistsvg . . . 25

4.3.1.2 Custom Symbols . . . 25

4.3.2 Yosys-JSON . . . 25

4.4 Data Structures . . . 28

4.4.1 Data Structure for the Diagram . . . 28

4.4.2 Symbol . . . 29

4.5 Development Environment . . . 30

4.5.1 Docker Container . . . 30

4.5.2 Building the Application . . . 30

4.5.3 Testing Environment . . . 31

4.5.4 Deployment of Application . . . 31

4.5.5 Continuous Integration . . . 31

5 Reading and Parsing of Files **31**

5.1 Reading of Yosys-JSON Files . . . 32

5.2 Parsing of Yosys-JSON Netlists . . . 32

5.2.1 Parsing of Objects . . . 32

5.2.2 Creation of Bus Splitters and Joiners . . . 33

5.2.2.1 Custom Algorithm . . . 34

5.2.2.2 Netlistsvg Algorithm 37
5.2.3 Connection of Objects 39
5.3 Symbols 39
5.3.1 Reading in symbols 39
5.3.2 Parsing symbols 40

6 Placing Components and Lines 40
6.1 Assignment of Symbols 41
6.1.1 Creation of Bus and Generic Symbols 41
6.2 Calculation of Positions 42
6.2.1 Converting to Cola Graph 42
6.2.1.1 Standalone Ports and Nodes 42
6.2.1.2 Paths 44
6.2.2 Calculation of Routing Parameters 44
6.2.3 Routing of Positions 47
6.3 Calculation of Lines 47
6.3.1 Conversion to the `libavoid` format 47
6.3.1.1 Conversion of Cola Rectangles 48
6.3.1.2 Conversion of Edges 48
6.3.2 Configuration of `libavoid` 48
6.3.3 Improved Performance of `libavoid` 49
6.3.4 Routing of Lines 50

7 Visualization and User Interface 50
7.1 Overview over Graphical User Interface (GUI) Features 50
7.2 Converting to Qt Objects 51
7.2.1 Nodes and Standalone Ports 52
7.2.2 Paths 52
7.3 Rendering Names 54
7.4 Visualizing the Schematic 54
7.5 Settings 55
7.5.1 Settings for Symbols 55
7.5.2 Settings for Routing 56
7.6 Navigation for Submodules 56
7.7 Functionality of the User Interface 57
7.7.1 Loading of Netlists 57
7.7.2 Zooming and Panning 58
7.7.3 Highlighting/Selection of Objects 58
7.7.4 Display of Additional Information 59
7.7.5 Loading Examples 60

        7.7.6 Export of Schematics 60
    7.8 Command Line Arguments 61
    7.9 Peer Test of the Application 61

8 Results **63**
    8.1 Runtime 63
        8.1.1 Setup 63
            8.1.1.1 Design: `regfile` 65
            8.1.1.2 Design: `VISCY-Controller` 65
            8.1.1.3 Design: `wb_tx` 65
            8.1.1.4 Design: `PicoNut-Controller` 66
            8.1.1.5 Design: `wb_rx` 66
        8.1.2 Results 66
            8.1.2.1 Unmodified `libavoid` 67
            8.1.2.2 `libavoid` with modifications 68
            8.1.2.3 `libavoid` in WebAssembly 69
            8.1.2.4 Graphs of the Messurements 69
        8.1.3 Conclusion 71
    8.2 WebAssembly Integration 74
        8.2.1 Setup 74
        8.2.2 Results 74
            8.2.2.1 Exceptions 74
            8.2.2.2 Context Menus 75
            8.2.2.3 Routing 75
        8.2.3 Conclusion 76

9 Conclusion **76**
    9.1 Summary 76
    9.2 Outlook 77

Bibliography **79**

A Appendix **a**
    A.1 `libavoid` Patch . . . . a
    A.2 Screenshots of Used Diagrams . . . . d

List of Abbreviations

ALU	Arithmetic Logic Unit
CD	Continuous Deployment
CI	Continuous Integration
CI/CD	Continuous Integration and Continuous Deployment
CLI	Command Line Interface
cola	constraint layout
EDIF	Electronic Design Interchange Format
EES	Efficient Embedded Systems
ELK	Eclipse Layout Kernel
EPL	Eclipse Public License
FPGA	Field Programmable Gate Array
GPL	General Public License
GUI	Graphical User Interface
HDL	Hardware Description Language
JSON	JavaScript Object Notation
LGPL	Lesser General Public License
moc	meta object compiler
OGDF	Open Graph Drawing Framework
rcc	resource compiler
RTL	Register Transfer Level
THA	Technische Hochschule Augsburg
UART	Universal Asynchronous Receiver/Transmitter
uic	user interface compiler
VHDL	VHSIC Hardware Description Language
VISCY	Very Reduced Interactive Set Cpmputer System
vpsc	Variable Placement with Separation Constraints
W3C	World Wide Web Consortium

WASM	WebAssembly

List of Figures

3.1 yosys show Example . . . 17
3.2 netlistsvg Example . . . 18
3.3 TerosHDL Example . . . 19
3.4 digitaljs Example . . . 21

4.1 Components of the Application . . . 22
4.2 Class Diagram of the Diagram Data Structure . . . 28
4.3 Class Diagram of the Symbol Data Structure . . . 29

5.1 Examples of Objects used in Schematics . . . 34
5.2 Creation of a bus joiner with netlistsvg Algorithm 1 . . . 37
5.3 Creation of a bus joiner with netlistsvg Algorithm 2 . . . 38
5.4 Creation of a bus splitter with netlistsvg Algorithm 1 . . . 38
5.5 Creation of a bus splitter with netlistsvg Algorithm 2 . . . 38

6.1 Example of a Input Port as a cola Graph Object . . . 43
6.2 Graph of Experiment Results . . . 46

7.1 Example of Points Generated by the Routing Library . . . 53
7.2 Example of a Path with Diverging Point . . . 53
7.3 Example of a Schematic Displayed in the Qt GUI . . . 55
7.4 Settings Dialog . . . 55
7.5 Hierarchy Browser . . . 56
7.6 Tabs for Submodules . . . 57
7.7 Problem with Selection of Paths . . . 59
7.8 Properties Dialog Example . . . 60

8.1 Graph of `libcola` Runtimes . . . 70
8.2 Graph of `libavoid` Runtimes . . . 70
8.3 Graph of Overall Runtimes . . . 71
8.4 Comparison of routing WASM and native . . . 75

A.1 Regfile Screenshot . . . d
A.2 VISCY-Controller Screenshot . . . e
A.3 WB-TX Screenshot . . . f

A.4 Pico-Controller Screenshot . . . . g
A.5 WB-RX Screenshot . . . . h

List of Tables

4.1 Additional SVG Attributes for Symbols 24
4.2 Additional SVG Attributes for Ports 24

5.1 Steps of the Custom Algorithm to Connect the objects in the Diagram. 36

6.1 Routing Parameters 44
6.2 Routing Parameters from Experiment 45

7.1 Feedback from Peer Test 63

8.1 Design `regfile` Indicators 65
8.2 Design `VISCY-Controller` Indicators 65
8.3 Design `wb_tx` Indicators 65
8.4 Design `PicoNut-Controller` Indicators 66
8.5 Design `wb_rx` Indicators 66
8.6 Unmodifided `libavoid`: cola Runtime 67
8.7 Unmodifided `libavoid`: avoid Runtime 67
8.8 Unmodifided `libavoid`: complete Runtime 67
8.9 Modified `libavoid`: cola Runtime 68
8.10 Modified `libavoid`: avoid Runtime 68
8.11 Modified `libavoid`: complete Runtime 68
8.12 Modified WebAssembly `libavoid`: cola Runtime 69
8.13 Modified WebAssembly `libavoid`: avoid Runtime 69
8.14 Modified WebAssembly `libavoid`: complete Runtime 69
8.15 Speedup of modified `libavoid` Compared to Unmodified 73

Listings

4.1 Example of an AND Gate Symbol 24
4.2 Instance of a Port in Yosys-JSON 26
4.3 Instance of a Cell in Yosys-JSON 27
4.4 Instance of a Netname in Yosys-JSON 27

6.1 Code for Input Port in libcola 42
6.2 Unmodified canFollow Functions 49
6.3 Modified canFollow Functions 49

8.1 Native `libavoid` Warning 76
8.2 WASM `libavoid` Warning 76

A.1 Patch for `libavoid` . . . . a

1 Introduction

1.1 Motivation

Industry-standard tools for Field Programmable Gate Array (FPGA) design from companies like AMD (Xilinx), Intel (Altera), and Lattice Semiconductor all provide ways to visualize digital circuits as interactive schematics [7, 25, 31]. This enables developers of digital circuits to gain an overview of the circuit. All offerings from these companies are closed source and are often integrated into large software suites. In contrast, the open-source toolchain OSS CAD Suite provides visualization through generated Graphviz files or third-party tools such as netlistsvg, which generate static SVG images [36, 61, 63].

The lack of interactive schematic viewers was also noted by the Efficient Embedded Systems (EES) group. The EES group is a research group at the ”Technische Hochschule Augsburg” (THA) that focuses on FPGA development for image processing, and also works on heterogeneous systems and RISC-V processors [22]. One of their projects, called PicoNut, aims to create a flexible, open-source RISC-V processor [23]. This also implies that the hardware and the toolchain used in development should be open, which led to the decision to use ECP5 FPGAs from Lattice Semiconductor and the OSS CAD Suite. Projects developed by the group are used in education, where students are able to develop their own extensions for the projects in order to learn hardware design. For this use case, it is important to have a schematic viewer that is easy to use and allows users to interact with the schematic, similar to the tools provided by AMD and Intel. This includes the ability to view submodules of the design without the need to generate a new schematic for each module, good labeling for the wires and components, and the ability to run the tool in a web browser so that students can use the software without installing anything. Additionally, the tool should support multiple Hardware Description Languages (HDL), because both Verilog and VHDL are used in the EES group. To find such tools, the OSS CAD Suite community was consulted in their Slack channel to ask if any existing tools are known. The result was that some tools for viewing schematics exist. One is the tool netlistsvg, which only generates static SVG images of the schematic. It is able to run in a web browser but does not allow viewing submodules without generating a new diagram [36]. Another tool is named digitaljs, and like netlistsvg, it can run in a web browser. It is mainly a simulator that displays the schematic of the design [10]. Though it is interactive and supports viewing submodules, it does not support VHDL designs out of the box. The Slack channel conversation indicated support for the idea of creating a new interactive schematic viewer tool that provides more interactivity than the existing tools [8].

The conversation in the OSS CAD Suite Slack channel and the lack of other solutions that meet the requirements led the EES group and the author of this thesis to the decision to develop an interactive schematic viewer.

1.2 Purpose of the Work

The aim of this work is to create an interactive schematic viewer for digital designs that allows the user to visualize a design. The project should be open source and compatible with the OSS CAD Suite. It must be able to run in a web browser. This is especially useful for students during lab exercises and lectures, where they can quickly view a design without the need to install any software.

To achieve this goal, the program needs to define internal data structures that represent the design with its components, connections, and structures. Additionally, the data presented in a netlist format needs to be parsed into the internal data structures for further processing. This is necessary for the program to read the types, names, and connections of components and modules.

Furthermore, it is required for the application to calculate the positions of individual components and the connecting wires/paths between them. This is necessary to generate a readable schematic that the user can understand. This step can be performed using existing routing libraries that apply mathematical algorithms to determine the positions of objects in the diagram.

The program needs to provide a user interface that allows the user to manipulate the diagram with certain actions. These include zooming, highlighting, and the ability to display additional information about components and wires/paths. This enables the user to gain a better overview of the design and understand how the circuit works.

Configurability is also an important aspect of the application. Users should be able to adjust the displayed diagram to suit their requirements. This can include the ability to change the icons of components and to modify settings of the routing library.

Another important aspect of this project is the ability to navigate into submodules of a design. This is especially useful when displaying larger designs that are split into multiple modules, so the user does not have to start the tool for each module separately. This adds flexibility, as the user does not have to decide beforehand which module of the design to open.

To have a better overview of the submodules when navigating through the design, a hierarchy browser is needed. This allows the user to understand the structure of the design and how the modules are connected to each other, which is especially important in larger diagrams with multiple layers of submodules. The browser should be interactive so that the user can click on a module to navigate into it.

The application should have good performance when viewing larger designs. This is important to ensure that the user experience is not hindered by long loading or response times when interacting with the program.

1.3 Outline of the Thesis

In the Chapter 2 ”Fundamentals”, the theoretical foundation of this work is laid out. This includes information about file formats, graphical libraries, and the development environment used. Chapter 3 ”State of the Art” discusses existing routing libraries, their features, and limitations. Furthermore, existing schematic viewers are presented and compared against the features required of an interactive schematic viewer. The next Chapter, 4 ”Overview”, gives a general overview of the structure. Additionally, it discusses the libraries and file formats used in the development of an interactive schematic viewer for digital circuits, as well as the development environment used. After giving an overview of the project, Chapter 5 ”Reading and Parsing of Files” discusses parsing files that are needed for the program to work. This includes the Yosys-JSON file format, which contains the netlists of the circuit, and the symbol files, which contain the symbols used for the components of the schematic. Chapter 6 ”Placing Components and Lines” describes how the positions of the components are calculated. This includes the assignment of symbols. Chapter 7 ”Visualization and User Interface” gives an insight into how the data structures are graphically displayed. Additionally, the requirements for the user interface and the ways in which the user can interact with it are addressed. Furthermore, the results of a peer test are presented. In Chapter 8 ”Results”, the results are presented. This includes performance measurements for the runtime. Furthermore, the integration of WebAssembly into the project is discussed. The last Chapter, 9 ”Conclusion”, gives a summary of the work and also discusses the next steps that can be taken to improve and expand the project.

2 Fundamentals

2.1 File Formats for Circuit Representation

For the representation of circuits, different file formats are in use [17, 24, 65]. In this chapter, the focus is on formats that are used to represent the structure of a circuit. The following formats are considered:

- EDIF
- VHDL
- Yosys-JSON

2.1.1 The Electronic Design Interchange Format (EDIF)

The Electronic Design Interchange Format (EDIF) is a format that looks similar to the LISP programming language because of its use of a notation with prefixes enclosed in parentheses. This allows the format to be easily read by machines and humans [40]. The format is not specified by a single company or tied to a specific tool, which makes it suitable for the exchange of circuit designs between different CAD systems and for communication with manufacturing equipment [17, 40]. It is possible for an EDIF file to contain the design with all the high-level information, which mainly consists of netlists. Since the introduction of the format, several versions have been released. The most recent version is EDIF 4.0.0 [17].

2.1.2 The VHSIC Hardware Description Language (VHDL) Format

The VHSIC Hardware Description Language (VHDL) is a standardized language used for modeling digital circuits. The language can be used in all phases of the design process. Like the EDIF format, it is readable by both humans and machines. The current version of the standard is IEEE 1076-2008 [24]. The VHDL language uses architecture blocks to indicate which type of description is being used for the design. There are multiple types of architectures, such as behavioral, dataflow, and structural. For the representation of the structure of a circuit, the `STRUCTURE` keyword is used. This structural description is used by the `Aliance` tools to model the netlist of a design [29].

2.1.3 The Yosys Open Synthesis Suite (Yosys) Netlist Format

The Yosys Open Synthesis Suite's (Yosys) netlist format uses JavaScript Object Notation to represent a circuit's netlist. The format is used by the Yosys synthesis tool. This open-source tool can synthesize a design from different hardware description languages, such as VHDL or Verilog, into a netlist. It can create multiple types of netlists, such as those containing only general logic blocks and others specific to a certain technology, so they can be used for further processing like place-and-route [64, 65]. The possibility to create netlist representations for different Hardware Description Languages (HDLs) and technologies makes the format suitable for the creation of a schematic viewer. For this, the Yosys tool can be used as a converter from different HDLs to the Yosys-JSON format. For example, a RISC-V core can be synthesized with Yosys, and the resulting netlist can then be converted to the Yosys-JSON format and viewed inside a digital schematic viewer.

2.2 Open Source Synthesis Toolchain

The OSS CAD Suite is a collection of open-source tools that allow the design of digital circuits. It contains tools for synthesis, place and route, FPGA programming, and additional utilities. It also supports different HDL languages like VHDL, SystemVerilog, and Verilog. This includes tools like Yosys, nextpnr, and GHDL. The suite is available for different operating systems as a prebuilt binary package and also provides a docker container for development [61].

2.3 The WebAssembly Standard

WebAssembly (WASM) is an open web standard defined by the World Wide Web Consortium (W3C) [60]. It provides a binary format for instructions that is executed in a stack-based virtual machine [59]. It allows applications with high performance requirements to run inside a web browser. While it is not focused solely on web-specific features, it also allows execution in non-web environments [60]. A WebAssembly application can run in parallel with JavaScript code. This allows a program to take advantage of the performance of WebAssembly while still using the features of JavaScript [34]. It is also possible to load WASM modules through a JavaScript API. WebAssembly code can be generated from different programming languages like C, C++, Rust, and others. This allows native code to run in a web browser [34]. For projects like the schematic viewer, WebAssembly can be used to provide a way to run applications written in languages like C++ or Rust in the web browser, enabling users to work with the application without the need to install it on their system.

2.3.1 The WebAssembly Compiler emscripten

The emscripten toolchain provides a way to compile C and C++ code to WebAssembly. It uses the LLVM compiler to create WASM applications from almost any C or C++ codebase, this includes Qt applications. It also supports graphics and sound APIs such as OpenGL and SDL2, and provides ways to interact with the filesystem [18, 20].

2.4 Graphical User Interfaces

Graphical User Interfaces (GUIs) provide an easy way for end-users to interact with an application. They can interact with the information displayed on the screen directly using a mouse, keyboard, and other input devices. This is different from Command Line Interface (CLI) programs, where the user has to type in commands with the correct syntax to interact with them [26, 41].

2.4.1 Qt Graphics Framework

The Qt framework is a cross-platform solution that allows the development of applications for different operating systems and platforms, such as embedded devices, desktop computers, mobile devices, and the web [39]. This allows a Qt program to run on systems like Linux, Windows, Android, and others with little to no changes to the codebase. The framework uses the C++ programming language with additional features, such as the signals and slots mechanism, that allow different Qt objects to communicate with each other [56]. All these features are provided by the meta-object system, which uses the Meta Object Compiler (MOC) as a preprocessor to convert the extended Qt-C++ code to standard C++ code [39]. There are additional tools and files that extend the functionality of the framework. For example, there are so-called `.ui` files that contain a Qt Widget layout of the application. These can either be loaded at runtime or, with the help of the UI Compiler (uic), be converted to a header file that can be included in the application [58]. Another extension of standard C++ is the Qt Resource System, which is a way to include additional files such as images, icons, and translations into an application. This has the advantage that it does not use system-specific methods to package the resources. In order for this to work, an additional file with the extension `.qrc` is used to define the prefix the files use and the path to the files. These files are processed by the Resource Compiler (rcc) to create `.cpp` files that contain the resources so they can be compiled into the application [57]. For the compilation of Qt programs, the `qmake` build system can be used, which provides a cross-platform wrapper around the native build tools. Since version 4 of Qt, the CMake build system is also a supported way to build Qt applications, it also allows the packaging of applications [39]. The cross-platform nature of Qt makes the framework suitable for the development of the schematic viewer, which should run on as many systems as possible, since the code can be written once and then used on different systems, and even on the web with the help of WebAssembly.

2.4.2 WebAssembly and Qt

The WebAssembly (WASM) support of Qt allows the development of applications that can run in the web browser. This can be used when the program does not require full access to the functions of the system it is running on. In order for a Qt application to be compiled to WebAssembly, the emscripten toolchain is required. For each release of Qt, a specific version of the `emsdk` is needed in order to successfully build the program. Additionally, the Qt libraries need to be compiled for WebAssembly in order to compile the application. At the time of writing, the WebAssembly version of the framework does not support all of the features available in the native version of Qt. Additionally, functionality such as exceptions is disabled by default. This means that if a developer needs to use such features, they have to compile a version of the Qt libraries that supports them. Furthermore, filesystem access is restricted to specific functions that allow the application to read and write files to the device's filesystem [54].

2.5 Development Environment

This section describes the different components of the development environment used to create the schematic viewer. These include containerization software, the testing environment, continuous integration, and the AppImage format, which can be used to distribute the application.

2.5.1 Continuous Integration

In order to ensure that the software works as expected and that changes made to the codebase do not break the functionality of the software, a continuous integration (CI) workflow is used. This allows the developer to run tests on their code changes before merging them into the main branch. Additionally, CI allows the developer to build and package the software using so-called continuous deployment (CD) workflows. There are multiple frameworks that can be used to create a CI/CD workflow, such as Jenkins, which will be discussed in the following section [38].

2.5.1.1 Jenkins Automation Server

The Jenkins project is an open-source project that provides an automation server capable of performing a variety of tasks. This includes building, testing, and deploying software. The project is written in Java and supports plugins that extend its basic functionality [28, 38].

2.5.2 Docker Container

The Docker project provides a way to create isolated environments that allow packaging or running an application without affecting the host system. These units are called containers. They allow the developer to create a reproducible environment that does not depend on the libraries and programs installed on the Docker host. These properties allow the user to run multiple containers on one host without them interfering with one another. Furthermore, a standardized container allows multiple developers to work on the same project while being confident that the software will behave the same for all of them. This is also useful for CI/CD workflows that package or test the software to ensure consistent results [11].

2.5.3 Testing Environment for Qt

When testing Qt applications, the Qt Test framework can be used. This is a unit test framework provided with Qt. It offers all the features of a test platform, with additional functions that allow the developer to test the GUI of the application. It is also integrated into Qt's build systems so that the tests can be compiled with them. When using CMake, the built executables can be run with the `ctest` command. There are also functions that run at specific stages of the test cases, allowing the developer to set up the environment before running the tests and to clean up afterwards. The results can be exported in multiple formats, such as xml, csv, or txt [55].

2.5.4 Distribution with AppImage

The AppImage format allows developers to package their applications for Linux systems. It has multiple advantages over other packaging formats. For one, it contains all the dependencies that the application needs to run [44]. This means that it can run on different distributions without needing to repackage the application, so the user does not depend on the distribution to update the package. The resulting package is a single file that can be run by the user. This also allows multiple versions of the same application to exist on the system [44, 45, 46].

For the developer, the AppImage format provides a way to package the application independently of the distribution. This means that the developer can provide the most up-to-date version of the application to the user [45]. The packaging step can be done by tools like `linuxdeploy`, which themselves are AppImage files. They can use a so-called “AppDir” that contains files like the executable, the program icon, and a desktop file that contains information about the application. When an “AppDir” with the mentioned files exists, it will be populated with the dependencies of the program. The tool then creates the AppImage file that can be distributed to the users of the program. Alternatively, the tool can also create an “AppDir” folder when one does not already exist and the correct files are given as parameters. The `linuxdeploy` tool also allows the use of plugins that add functionality to the packaging process. For example, there is a Qt plugin that helps when creating AppImages that contain a Qt application [47].

3 State of the Art

3.1 Routing Libraries

Lines and objects in the schematic need to be routed in order for the displayed schematic to be readable. This step can be performed manually, but there are multiple libraries that provide functionality for routing lines and objects.

For the creation of a digital schematic viewer, it is important that the library used satisfies certain requirements.

- The library needs to be usable in C++ because the project will be written in this language in order to use the functionality of the Qt framework to create WASM applications. This ensures easier integration of the library and avoids unexpected behavior when using two different WASM compilers.
- The library needs to be able to define ports on nodes. This is important for digital schematics because symbols in such a graph have connectors for signals at specific positions.
- The library needs to provide configuration options for how the routing should be performed. This is important because schematic routing needs to be done in a way that is readable and understandable. It should be possible to configure the distances between nodes and the distances between edges, and it should allow edges to be merged when they originate from the same port to avoid unnecessary lines in the schematic. Additionally, line routing should support orthogonal edges for better readability and allow setting the distance before the first bend of a line.
- The project will be licensed under the GPL3 license, and the library needs to be compatible with this license.
- The library should have good performance and scalability. This means that the runtime should be low and should not increase significantly with the size and complexity of the design. This is important so that the user does not have to wait a long time to see the schematic.

3.1.1 `elk/elkjs`

The Eclipse Layout Kernel (ELK) is a Java library that provides various layout algorithms and a way to integrate these algorithms into viewers and editors of diagrams [16]. It consists of three parts, which the project calls layers. The first layer is the ”Basic Layer”, which provides the data structures and algorithms. The second layer is the ”Plain Java Layer”, which provides the layout engine capable of routing graphs provided in the so-called `ElkGraph` format. This layer also provides the metadata service, which contains all the supported algorithms and their properties. The third layer is the ”Eclipse Layer”, which can be used if the project creates an Eclipse application [15].

The `ElkGraph` format contains various objects and data structures. Some of these are nodes, edges, and ports. Nodes are blocks that are connected by edges. They can also have ports, which define the connection points of the edges. This means that the ELK library satisfies the requirement that a routing library used in this project must be able to define connection points at specific positions on the nodes [14].

The ELK library also provides layout and spacing options that can be used to configure the output after routing [15]. This is especially true for the so-called ”layered” layout algorithm, which is used in the netlistsvg project, as discussed in Chapter 3.2.2. According to the documentation, it allows the user to configure the distance between nodes and edges. It also allows the merging of edges that originate from the same port and the specification of specific port positions. When the routing type is set to ”orthogonal,” it also allows setting the distance from the node to the first bend of a line [13]. This means that the ELK library satisfies the requirement that the library should provide configuration options.

The language used in the ELK library is Java, which is not compatible with the project because the project will be written in C++ [16]. The `elkjs` library, which is an implementation of the ELK library in JavaScript, is also not compatible with the project [30].

Another point that would make the integration of the ELK or `elkjs` library difficult is the license under which these projects are published. Both projects are licensed under the Eclipse Public License 2.0, which, according to the FAQ, is not directly compatible with the GPL3 license, under which the project created in this thesis will be published [12].

Lastly, the ELK library has good performance and scalability. This can be seen in the netlistsvg project, described in Chapter 3.2.2. When using the program with larger diagrams, such as those used in Chapter 8.1, the performance remains good, with the runtime being only a few seconds even for larger designs.

While the ELK library satisfies some of the requirements important for use in this project, it is not compatible with the project due to the programming language and license used.

3.1.2 Open Graph Drawing Framework

The Open Graph Drawing Framework (OGDF) is another library that provides routing and graph drawing algorithms. The OGDF is written in C++, which meets the requirement that the library should be in this language, making the creation of a

WASM application easier and more predictable. It also uses no additional libraries, which makes the integration into projects easier [9, 37].

The OGDF library, like the ELK library discussed in Chapter 3.1.1, has multiple different layout algorithms that can be used to route graphs. However, in contrast to ELK, the OGDF library does not support the creation of fixed ports on nodes. This was determined by testing the library and examining the documentation [51]. Testing also revealed that OGDF supports exporting graphs as SVG files, which the ELK library does not, as documented on the ELK website [16].The missing support for fixed ports on nodes makes the OGDF library unsuitable for the project, as digital schematics require this feature. Testing also did not reveal any workarounds that could be used to create fixed ports on nodes while still using automatic layout with the provided algorithms.

Testing and studying the documentation revealed extensive configuration options for the routing algorithms. It includes multiple algorithms, each with its own configuration options. The ones tested during evaluation were ”PlanarizationLayout” and ”SugiyamaLayout,” in combination with ”OrthoLayout” for the edges. The ”PlanarizationLayout” or ”SugiyamaLayout” can be used to generate a layout of the nodes. The ”SugiyamaLayout” provides options to set the distance between nodes, whereas the ”PlanarizationLayout” does not. The ”OrthoLayout,” however, provides multiple options. For example, it allows setting the distance between edges and the margin around the graph [48, 49, 50]. This means that the OGDF library provides most of the desired configuration options. Some other algorithms in the library might be able to provide the missing ones.

The OGDF library is dual-licensed under the GPL2 and GPL3 licenses, which makes it compatible with the license under which the project will be published [9, 37, 51].

Due to the missing support for fixed ports on nodes, which is required when creating a schematic viewer, the performance of the library was only tested for small handwritten examples. Where the performance was in the range of milliseconds.

The OGDF library is a good candidate for the routing library, but the missing support for fixed ports on nodes makes it unsuitable for the project.

3.1.3 Adaptagrams Libraries

The Adaptagrams project is a collection of libraries that enable the creation of programs for generating adaptive diagrams. It is suitable for tools used in drawing, diagram layout, graph drawing, and other related tasks [6, 35]. The combination of multiple libraries from the collection can be used for routing in digital schematics.

All the libraries that are part of the Adaptagrams project are licensed under LGPL2.1. Combining this with code licensed under the GPL3 is possible as long as the license used is one released after GPL2 [21]. This makes it possible to use the Adaptagrams project in this project.

The libraries are written in C++, which makes integration into the project, even when using WASM, easier and more predictable. The project also does not require additional libraries, making it a cross-platform solution [6]. This satisfies one of the requirements set at the beginning of the chapter.

Some of the libraries provide functions that can export SVG files of the current state of the diagram, which helps with debugging. These functions are documented in the Doxygen documentation of the libraries [3, 4].

The performance of the libraries was tested with smaller handwritten examples and with the finished program. The results of the performance tests with the application can be seen in Chapter 8.1. The performance of the libraries is not comparable to that of the ELK library. When routing larger and more complex designs, the runtime increases significantly, reaching several minutes.

While reviewing the code of the libraries, it was discovered that they use exceptions, which requires building the Qt library from source to enable this feature for use in WebAssembly.

The libraries that can be used will be discussed in the following section.

3.1.3.1 `libvpsc`

The Variable Placement with Separation Constraints (VPSC) library helps with layout problems. For this purpose, it implements a solver that can calculate squared differences. These are calculated for placement vectors based on a configured separation in order to determine an optimal position [5, 6]. It is a library used by other components in the Adaptagrams project to perform routing.

3.1.3.2 `libcola`

The Constraint Layout (cola) library provides functionality to perform force-directed layout. This can be used to create graphs with no overlapping nodes and visible clusters of nodes. This means that the library can be used to calculate the positions of nodes in a graph without having to provide starting positions. The only requirement is to define the edges between the nodes and the constraints that the edges and nodes must satisfy. This can be seen in the examples provided in the library's repository

[35]. Even without native support for fixed ports on nodes, the library can create such ports using a workaround. For this to work, the node representing the symbol must be created with additional nodes placed at the positions where ports should be. These nodes are connected using edges. To fix them at the specified positions, constraints are applied. These are set up in such a way that the ports are not allowed to move. An example of this can be seen in Listing 6.1.

This means that libcola is able to meet the fixed port requirement defined at the beginning of the chapter, which is necessary for the project.

To create the diagrams, two different layout algorithms are provided: the deprecated ”ConstrainedMajorization” algorithm and the ”ConstrainedFDLayout” algorithm, where ”FD” stands for ”force-directed,” which uses gradient projection for its calculations. With the help of different types of constraints and cluster boxes, the algorithm is able to define the distance between nodes [2].

This makes it possible to create readable and understandable diagrams, which is a requirement for the project.

3.1.3.3 `libavoid`

The libavoid library provides functionality to route edges around nodes. This is necessary to create readable diagrams. It is required in addition to libcola, which is described in Chapter 3.1.3.2. The reason for this is that libcola only computes the positions of the nodes according to the given constraints but does not calculate the positions of the edges, unlike ELK (described in Chapter 3.1.1) and OGDF (described in Chapter 3.1.2).

The library uses a custom-written algorithm to route the edges [3]. It also provides multiple configuration options to control the routing. For example, it can be configured to use orthogonal routing for the edges. Furthermore, the distance between the node and the first bend, as well as the distance between edges, can be configured. It also allows the merging of edges that originate from the same port [1]. This means that the library, in combination with libcola, provides the needed configuration options.

3.1.4 Conclusion

The previous sections presented different routing libraries. All of the libraries offer many settings to control the routing results. elkjs and ELK are not usable in this project because they are written in Java and JavaScript, which are not compatible with the project. Additionally, the libraries are licensed under the EPL, which is not compatible with the GPL3 license under which the project will be published. The OGDF library does not have any issues with the license or the programming language used, but it does not support fixed ports on nodes, and no workaround for this limitation could be found. This makes it unusable in this project. The Adaptagrams libraries are written in C++ and are licensed under a GPL3-compatible license. The creation of fixed ports is also possible using a workaround with the libcola library. The performance of the libraries decreases significantly with the increasing size and complexity of the designs. Though the Adaptagrams libraries are not as performant as the ELK library, they are the only libraries that allow the creation of fixed ports on nodes and are compatible with the language and license used in this project. This means that, even though the Adaptagrams libraries are not the best-performing options available, they provide features that cannot be easily added to other libraries. This makes them the best choice for a first implementation of the schematic viewer.

3.2 Existing Schematic Viewers

Before starting this project, it was important to look at existing open source solutions for the creation of schematics for digital circuits. The goal was to find out if there were existing solutions that could meet the requirements for a schematic viewer that are set in Chapter 1.2. The requirements are:

- It needs to be usable in the web browser. This is useful for testing the program, as no installation is required. This is especially helpful when used in education.
- It needs to create readable and understandable schematics. This also means that signals must be labeled.
- The user needs to be able to interact with the schematic. This includes the ability to zoom, highlight, and display additional information about components in the schematic.
- It should be possible to customize the diagrams, for example by changing the symbols used.
- It should be possible to view the content of submodules of designs by clicking on them.

The following sections will present existing open source solutions and compare them with the requirements.

Note: To get comparable results, all the tools received a `json` or `verilog` file that was run through the synthesis tool Yosys using the `prep` command [62]. This performs a general synthesis which can be used to generate Register Transfer Level (RTL) representations of the design. The design used in this comparison was the implementation of an Arithmetic Logic Unit (ALU) that is part of a small 16-bit processor called Very Reduced Instruction Set Computer System (VISCY), which is used in education at the “Technische Hochschule Augsburg” (THA). The design is written in VHDL [29].

3.2.1 `yosys show`

This solution is a built-in feature of the Yosys synthesis tool. It is a Yosys command that can be directly called inside the Yosys shell, added to a Yosys call, or included in a Yosys script file. Because of this, the tool cannot be run natively in the web browser, which is a requirement for the schematic viewer as set in Chapter 3.2. It uses the Graphviz tool to generate schematics that represent the design loaded into the Yosys tool. The command can generate multiple file formats: `svg`, `ps`, and `dot`. Figure 3.1 shows the VISCY ALU RTL schematic generated by the `show` command. It generated a `dot` file, which was converted to `pdf` with the command `$ dot -Tpdf input.dot > output.pdf`.

As can be seen in Figure 3.1, the diagram’s signals overlap significantly, making it difficult to read. Additionally, the signals are not directly labeled, only the ports have names. It is important to note that when generating diagrams with named intermediate signals, shapes containing the signal names are used to insert the names, which congests the diagram even further. Another point is that due to the use of Graphviz, the diagrams only use shapes and lines provided by the library, which makes it harder to identify different types of blocks at a glance. These points show that the diagrams generated with the `show` command are not easily readable, which is a requirement for the schematic viewer set in Chapter 3.2. This also means that customizing diagrams with, for example, custom symbols is not possible, which is another requirement.

When using a dot viewer program like `xdot`, it is up to the user to interact with the diagram. This means that it is possible to zoom into the diagram and highlight certain signals and nodes. However, it is not possible to display additional information about the nodes beyond what is already shown. Therefore, this requirement, set in Chapter 3.2, is not completely satisfied.

Testing the command with larger diagrams containing submodules revealed that dot representations of the submodules are generated. However, viewers like `xdot` do not

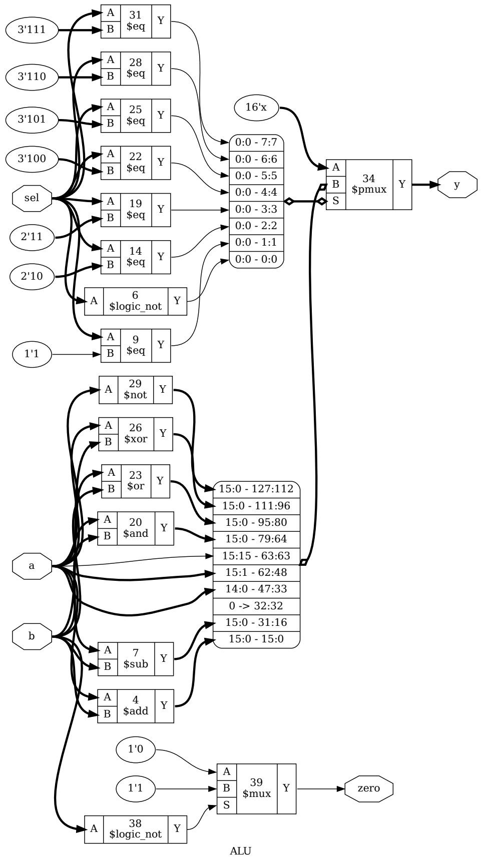


Figure 3.1: yosys show Example

support showing submodules if they are in the same file. To view them, the user needs to generate a separate diagram for each submodule or copy the different submodules into separate files. This limitation might be mitigated by using a different viewer that can handle multiple diagrams in one file. In its current state, this solution does not satisfy the requirement for a schematic viewer set set in Chapter 3.2.

3.2.2 netlistsvg

The netlistsvg project allows the user to generate schematics. For this, the tool uses netlists in the Yosys-JSON format and draws schematics into SVG files, which can then be viewed in, for example, a web browser. The project is written in TypeScript, which makes it easily integrable into a web application. This has already been done by the developer to provide a demo of the tool. This satisfies the requirement, set in Chapter 3.2, to have the possibility to run the schematic viewer in the web [36]. The following figure, 3.2, shows a schematic of the VISCY ALU generated by the netlistsvg tool.

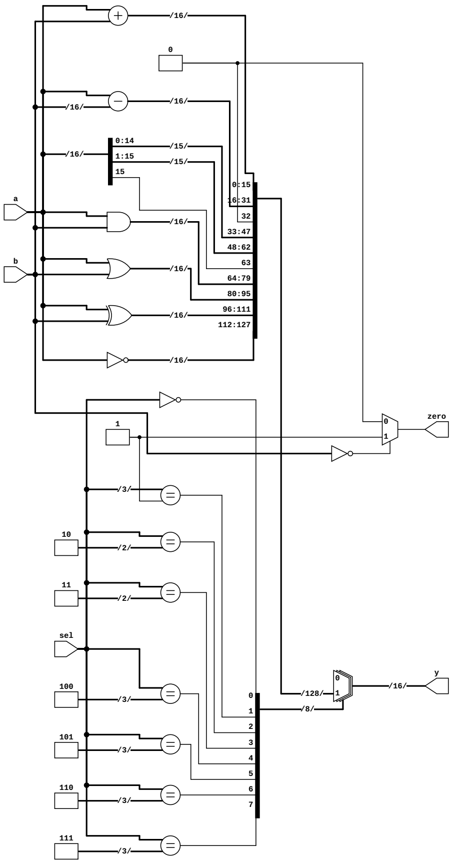


Figure 3.2: netlistsvg Example

Figure 3.2 shows that the diagram is quite readable. It uses distinct symbols for the different components, allowing them to be identified at a glance. It also tries to avoid overlapping signals, and even when they do overlap, the diagram remains understandable. However, the netlistsvg tool only shows the names of ports in the schematic; names of signals are not shown, which makes it difficult to identify which part of the diagram corresponds to a specific part of the HDL description, especially in larger diagrams. This means that the requirement for readable diagrams is met for smaller diagrams, but in larger diagrams the missing signal names make it hard to identify signals in the HDL source.

The netlistsvg tool only creates SVG files, which limits interaction with the schematic. It is possible to zoom and move the diagram, but highlighting of lines is not supported. Additionally, only the information depicted in the `svg` file is available, no additional information can be shown. This means that the requirement for interaction with the schematic is not met.

Netlistsvg allows the customization of symbols by using different `svg` files that contain custom symbols [36]. This means that the requirement for customization is met.

Because the tool only generates static `svg` files that contain the schematics, it is not possible to view the content of submodules. The only way to see the content of a submodule is to generate another `json` file and run the tool again. This means that the requirement for viewing submodules is not met.

3.2.3 TerosHDL

TerosHDL is a toolbox and IDE for HDL development that can be used inside VSCode and VSCodium. This includes a schematic viewer [42]. It was discovered that the schematic viewer in TerosHDL uses the netlistsvg tool to generate schematics [43]. This means that most of the points discussed in Chapter 3.2.2 also apply to the TerosHDL schematic viewer. Differences will be discussed in this section, with reference to Figure 3.3.

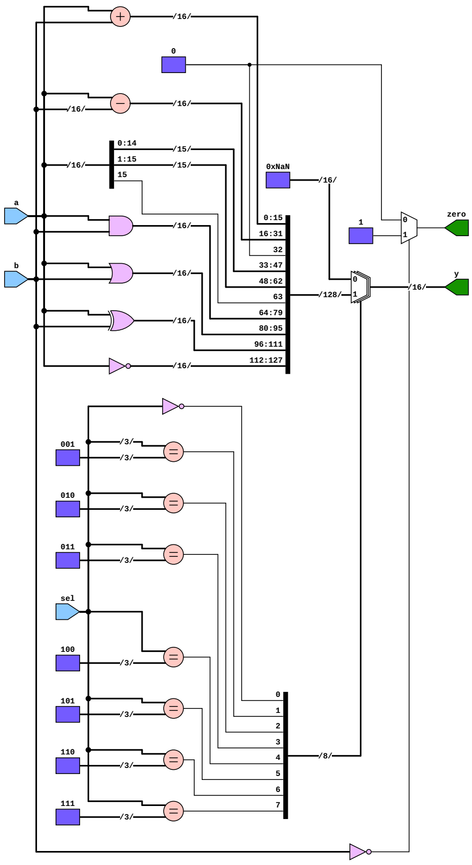


Figure 3.3: TerosHDL Example

Even though TerosHDL is written as a VSCode extension and the program is able to run inside the web browser, the schematic viewer is not usable in the browser. This is because it relies on local tools like Yosys to generate the Yosys-JSON files. This means that the requirement for running in the web browser is not met.

In contrast to the netlistsvg tool, the TerosHDL schematic viewer highlights the signals when hovering over them, allowing the user some interaction with the schematic. This means that the requirement for interaction is partially met.

Another point is that while testing the TerosHDL schematic viewer, no option was found to customize the symbols used in the schematic. The tool modifies the default netlistsvg symbols, as can be seen when comparing Figures 3.2 and 3.3, but the settings provide no option to change them to a custom `svg` file containing symbols. This would require modification of the source code to add this feature, meaning that the requirement for customization is not met in the current state of the tool.

3.2.4 digitaljs

The digitaljs project provides a simulator for digital circuits and also generates a schematic view of the loaded design. An online example was used to compare the project with the requirements set in Chapter 3.2 [10]. By providing this online example, it is shown that the project is usable in the web browser, satisfying the requirement set in Chapter 3.2. The tool loads designs using their HDL descriptions directly. However, it only supports SystemVerilog, Verilog, VerilogHex, and Lua scripts, which made it necessary to convert the VHDL design of the VISCY ALU to Verilog. This was done with the help of Yosys in combination with the GHDL plugin. The converted file was used to generate the schematic shown in Figure 3.4.

The schematics generated by the digitaljs project are quite readable. It uses symbols for each component, and lines do not overlap. It also names all the signals and logical blocks. This means that the requirement for readability is met.

Digitaljs provides ways to interact with the schematic by allowing zooming and moving the schematic. It also allows the highlighting of different signals and nodes in the diagram, although it only does this when hovering over the object. Like the other tools discussed, it does not provide a way to show additional information about the components. This means that the requirement for interaction is only partially met.

The tool also does not have an option to customize the symbols used, which means that the requirement for customization is not met.

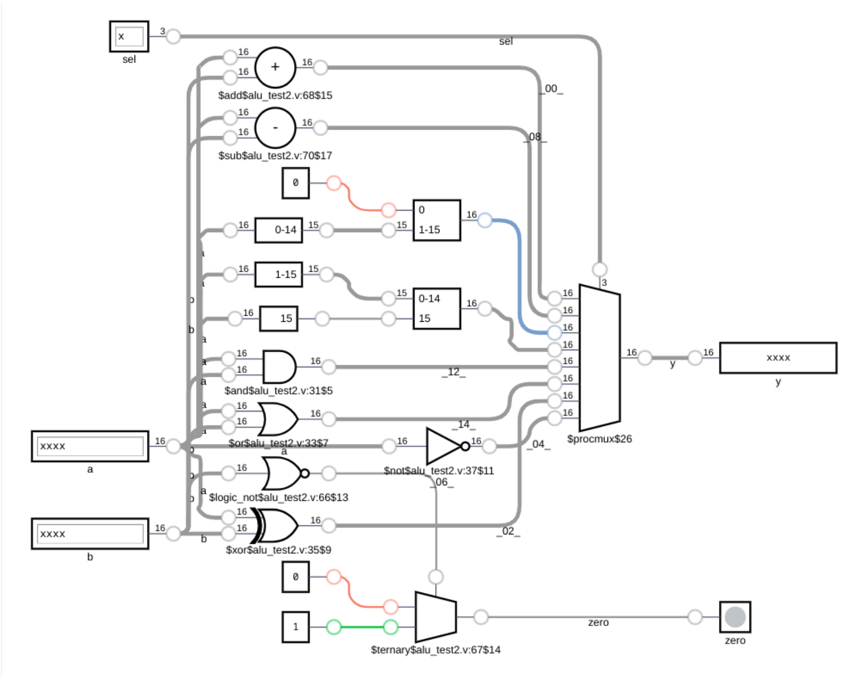


Figure 3.4: digitaljs Example

While testing the tool with diagrams of designs that include submodules, it was discovered that the tool allows the user to view the content of submodules. This means that the requirement for viewing submodules is met.

3.2.5 Conclusion

The previous sections analyzed various existing open-source solutions for the creation of schematics for digital circuits and compared them with the requirements set in Chapter 3.2. Although all the tools satisfy some of the requirements, none of them satisfy all of them. This means that one must either accept that some of the features mentioned in Chapters 3.2 and 1.2 are missing and use one of the existing solutions, modify an existing project, or create a new project that satisfies all the requirements. This led to the decision to create a new project that meets all the requirements. The following chapter will describe the design and development of the new schematic viewer.

4 Overview

This chapter will provide an overview of the project. It will explain the structure of the application and the libraries used for it. Additionally, the formats of the files processed by the application and the data structures used for storing information will be described. The development environment used for the project will also be explained.

4.1 Structure

In order to develop the schematic viewer, a structure is needed that contains multiple components responsible for specific tasks of the application and connects them together via interfaces. This allows for modularity when working on certain aspects, as it enables the developer to change one component without affecting the others.

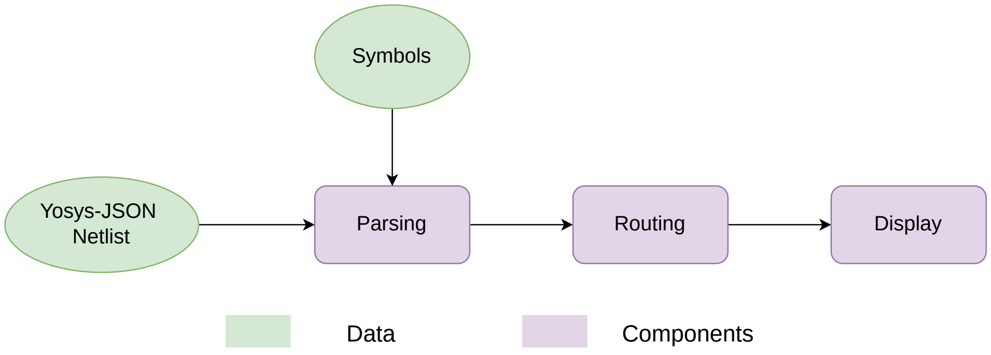


Figure 4.1: Components of the Application

As shown in Figure 4.1, the application consists of three components.

The **Parsing** component is responsible for reading the netlist of the digital circuit and converting it into internal data structures. This also includes the connections between different components. Additionally, the symbols used for the visualization of the components are parsed and stored within the application.

The **Routing** component is responsible for two tasks. The first task is to assign the symbols parsed in the previous step to the components of the netlist. If no symbol is found, it generates a default one. This is necessary because the second task of routing requires the size and position of ports to calculate the position of the components and paths/wires in a 2D space.

The **Display** component is the main Qt application and is responsible for multiple tasks. It converts the internal data structures into objects that the Qt framework can display. It also handles user input such as zooming, highlighting of objects, and the display of additional information about the objects. Furthermore, it allows the user to change settings that affect the display, such as the symbols or routing parameters.

4.2 Libraries

The application uses multiple libraries to accomplish its tasks. These libraries provide the graphical user interface, assist in parsing data, and offer algorithms for the routing of the components.

The Qt framework consists of multiple libraries that are called modules [52]. The program uses libraries that allow for the creation of a graphical user interface (GUI) using widgets. Additionally, the application uses modules that enable the handling of `JSON` and `XML` files. This is used for parsing the netlist and symbol data.

The project also uses multiple libraries from the Adaptagrams project for routing the components and paths/wires of the circuits. These include `libvpsc` as a requirement for other Adaptagrams libraries, `libcola` for creating the layout of the components on a 2D plane, and `libavoid` for calculating the position of the wires/paths between the components without overlapping with components or with each other. Static linking is used because it is more performant for WebAssembly (WASM) and it is more likely for unchanged code to compile successfully when using static linking [19].

4.3 File formats

4.3.1 Symbols

The symbols need to be replaceable by the user. This is achieved by providing them to the application as an SVG file, which has additional tags added to it to include information about the symbols that is important for the routing of components. This ensures that port positions and similar data are not hardcoded into the program but are changeable, allowing the user to create symbols of different sizes and shapes.

To achieve this, the existing format defined by netlistsvg is used as a base. The documentation for it can be found on the GitHub page of the project [36].

As can be seen in Listing 4.1, the SVG data used to describe the symbols uses normal SVG tags. In addition to that, there are tags that start with `s:` and contain information that is important for the router. Table 4.1 explains the functionality of the tags that are in the outer group tag.

Listing 4.1: Example of an AND Gate Symbol

```
<g s:type="and" transform="translate(150,50)" s:width="30" s:height="25">
  <s:alias val="$and" />
  <s:alias val="$logic_and" />
  <s:alias val="$_AND_" />

  <path d="M0,0 L0,25 L15,25 A15 12.5 0 0 0 15,0 Z" fill="#eebaff" class="$cell_id" />

  <g s:x="0" s:y="5" s:pid="A" />
  <g s:x="0" s:y="20" s:pid="B" />
  <g s:x="30" s:y="12.5" s:pid="Y" />
</g>
```

Attribute	**Description**
`s:type`	The type of the component. This is used to assign the symbol to a netlist component.
`s:width`	The width of the bounding box that contains the symbol. This is needed for routing.
`s:height`	The height of the bounding box that contains the symbol. This is needed for routing.
`s:alias`	An alias for the component. This is used as an alternative type when assigning the symbol to a netlist component.

Table 4.1: Additional SVG Attributes for Symbols

In addition to the tags used in the group tag, there are additional groups inside the symbols that contain the ports of a symbol. These groups also have additional tags that will be explained in Table 4.2.

Attribute	**Description**
`s:x`	The x position of the port relative to the symbol's coordinates.
`s:y`	The y position of the port relative to the symbol's coordinates.
`s:pid`	The port ID that is used to match the position of a port in the netlist to the one on a symbol.

Table 4.2: Additional SVG Attributes for Ports

4.3.1.1 Differences to netlistsvg

In contrast to the format used by netlistsvg, there are some limitations that result from the differences between the two projects. These differences are:

- SVG `text` tags are not used to give symbols a name. In this project, this is done directly by rendering labels with Qt objects.
- Creating two ports at the exact same position on a symbol is not supported, because this can lead to problems during routing.
- When defining colors inside the `svg` tag, it needs to be done separately for each type of shape to work around a limitation of the Qt SVG renderer.

4.3.1.2 Custom Symbols

It is also possible to create custom symbols for specific modules of a netlist. This requires the user to create a group like the one shown in Listing 4.1 with a custom shape. In order for the program to recognize the custom symbol, the user needs to set the type and alias to the type used in the netlist and additionally use the same port IDs as those used in the netlist. More information about this can be found in the user guide of the project.

4.3.2 Yosys-JSON

The program needs to be able to display digital circuit schematics that are written in different HDLs such as VHDL or Verilog. This requires the use of a netlist format that can be generated from these HDLs. For this, the Yosys-JSON format is used, which is independent of the language used to create the hardware design and can be synthesized by the Yosys synthesis tool. A description of the format can be found in the Yosys documentation [65]. It contains a lot of information, of which only some parts are needed to create the schematic. At the top level of a JSON file in this format, there is general information about the Yosys version used to create the file and an object called `modules`. This contains the different modules of a hardware design. These modules have an attributes object where only the field `top` is needed. If it is present, it means that the module is the highest in the hierarchy of the design. Other objects that are needed to create a schematic are the `ports`, `cells`, and `netnames` objects. The following listings only contain fields that are needed for the creation of a schematic. All additional fields can be looked up in the Yosys documentation [65].

The `ports` object contains further objects that describe each port of a module. These ports are used to connect to instances of other modules or serve as the inputs/outputs of the entire design. The name of each instance is the name of the object, which in the case of the example in Listing 4.2 is `cout_out`. Additionally, two other attributes are needed: the `direction`, which indicates the direction in which the signal flows through the port. This attribute can have three values: ”input”, ”output”, and ”inout”. The direction is interpreted from the perspective of the module, meaning a signal flowing into the module is declared as an ”input”. The second attribute, `bits`, contains numbers and strings, each representing one bit of the port. When the bits are numbers, this means that the bit with number n is connected to every other bit with the same number n. When the bits are the strings ”0” or ”1”, this indicates that the bit is connected to ground or to the supply voltage, making it a logical 1. Other values such as ”X” or ”Z” may exist but are generally not needed for creating a schematic. The entries are sorted with the least significant bit first, meaning the highest value bit is the last one in the array.

Listing 4.2: Instance of a Port in Yosys-JSON

```
"cout_out": {
    "direction": "output",
    "bits": [5, 6, 7, 8]
}
```

The `cells` object contains the description of individual components inside a module. These are primitives like AND gates or D flip-flops in RTL netlists but can also be primitives from a technology library if the netlist was synthesized for a specific one. An example of a `cell` can be seen in Listing 4.3. As with the `ports` object, the name of the object is the JSON instance name. Furthermore, there is a `type` attribute that specifies what kind of primitive the instance is. This can also be the name of another module, indicating that it is a submodule of the current one. The `connections` attribute holds information similar to the `bits` attribute of the `ports` object. The difference is that this object contains the `bits` vectors for all the inputs and outputs of the instance. The last object is `port_directions`, which, like the `bits` field, is similar to the `direction` field of the `ports` object. It defines objects representing the direction of the port for each one of them. These directions are from the perspective of the cell, meaning that an ”input” flows into the cell and an ”output” flows out of it.

Listing 4.3: Instance of a Cell in Yosys-JSON

```
"$procmux$7": {
    "type": "$mux",
    "port_directions": {
      "A": "input",
      "B": "input",
      "S": "input",
      "Y": "output"
    },
    "connections": {
      "A": [9, 10, 11, 12],
      "B": ["0", "0", "0", "0"],
      "S": [3],
      "Y": [17, 18, 19, 20]
    }
  }
}
```

The `netnames` object contains the names of certain bit groups in the module. An example can be seen in Listing 4.4. As with the other objects, the name of the JSON object is the name of the group. These names represent the signals inside the HDL description and are useful for labeling the wires in the schematic. The `hide_name` attribute indicates whether the name should be hidden or not. From studying multiple netlists, it could be determined that this attribute is set to 1 when no real name was provided in the HDL description, meaning that it was generated by Yosys. The `bits` attribute functions like the one in a `ports` object, meaning that it contains the numbers of the bits that are part of this signal. Another field that is important is the `unused_bits` attribute. This one is not documented in the Yosys documentation, but it can be seen in generated netlists. It is a string that contains the indices of the bits in the `netname` that are not connected to anything.

Listing 4.4: Instance of a Netname in Yosys-JSON

```
"counter_reg": {
    "hide_name": 0,
    "bits": [9, 10, 11, 12],
    "attributes": {
      "unused_bits": "2 3"
    }
  },
```

4.4 Data Structures

4.4.1 Data Structure for the Diagram

The application needs to define data structures to store the information parsed from the netlist. These must contain all the information required to create the schematic. A higher-level overview is shown in Figure 4.2.

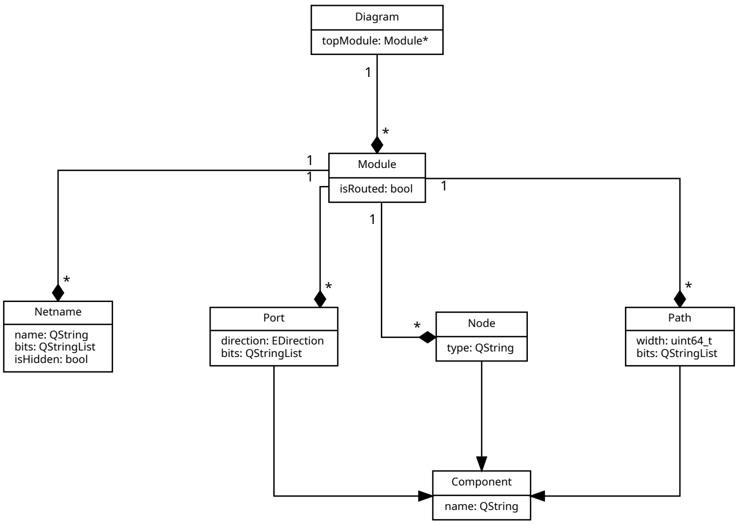


Figure 4.2: Class Diagram of the Diagram Data Structure

The UML diagram only contains the most important attributes of the objects and omits the methods inside the classes entirely. Additionally, some relationships between classes are not drawn in Figure 4.2 to improve readability.

A `Diagram` class is the data structure that holds all the data of one netlist. It contains the `Module` objects of the netlist and also defines which module is the top module of the design.

The `Module` class contains the different data structures that were discussed in Chapter 4.3.2. Additionally, it also contains the `Path` objects that describe the connections between the different components.

The `Netname` class contains the data of the `netnames` objects of the netlist. This means it contains the names of the different signals that are part of the module.

The `Path` class has no representation inside the netlist. It models the connection between `Port` and `Node` objects. For this, it contains additional references to the `Port` objects of the components.

The `Node` class is used to represent the different components of the netlist, where they are named `Cell`. In addition to the type of the cell, it also contains instances of `Port` objects that represent the inputs and outputs of the cell.

The `Port` class is used to represent ports of cells and also the ports of `Modules` that are defined in the netlist. In addition to the attributes shown in Figure 4.2, it also contains a pointer to the connected `Path` object. If it is part of a cell, a reference to the `Node` object is also present.

The three classes `Cell`, `Port`, and `Path` are the data structures that contain the data drawn in the schematic. All of them also inherit from the `Component` class, which contains common data such as the name.

4.4.2 Symbol

The application needs to save the information about the graphical symbols with their ports and sizes in order to display the objects of the schematic correctly. A representation of the data structure used for this is shown in Figure 4.3.

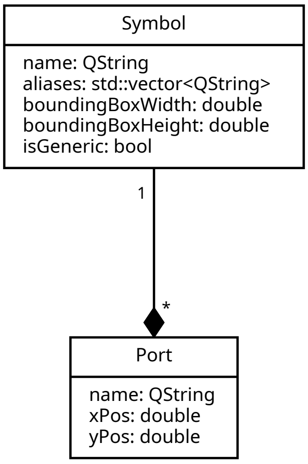


Figure 4.3: Class Diagram of the Symbol Data Structure

The `Symbol` class is the main class that contains all the information about a symbol. It contains instances of the `Port` class that represent the ports of the symbol to identify where they are placed and what position they have relative to the symbol. Additionally, the `Symbol` class stores the size of the symbol, the name, and alias names as specified by the symbol format discussed in Chapter 4.3.1. The `isGeneric` attribute provides additional information. It specifies whether the symbol is a generated generic symbol or not. A `Symbol` instance also stores the SVG representation of the symbol as a string. This is used to generate the graphical representation of the symbol.

4.5 Development Environment

4.5.1 Docker Container

The application is developed inside a Docker container. This is done to ensure consistent results when building the application. Since the routing libraries from Adaptagrams use exceptions, it is necessary to build the Qt project from source during the Docker build process. The build process of the Docker container is described in the `Dockerfile` located in the project's repository. It installs all the required packages for building Qt and this project's documentation. Building the Qt framework requires two separate builds: one that builds the Qt framework for the host system and another that cross-compiles the framework for the WebAssembly target. At the time of writing, the Docker container is based on the `debian:bookworm` image. Building from source is necessary because the application and the routing libraries use exceptions. To enable exceptions in the build of the WASM Qt libraries, the flag `--fWASM-exceptions` must be set [54].

For using the Docker container, the `docker-compose` tool is employed. This allows the container to be configured with a single command. The configuration is saved in a file. For this project, the graphical display and the location of the Qt installation are configured.

More information about building and using the Docker container can be found in the project's user guide.

4.5.2 Building the Application

The build system of the application is designed to be easy to use. This allows the developer to build the application without having to deal with the configuration manually. For this purpose, a script is provided that handles the configuration of the build system. The only requirements are the dependencies of the application and a variable that points to the desired Qt installation directory. More information about this can be found in the project's user guide.

4.5.3 Testing Environment

The application needs to be tested to ensure that it functions correctly. For this purpose, a testing environment is provided. To enable testing of the Qt application, the `QtTest` module is used. This module provides a framework for testing Qt applications [55]. It is configured to work with the build system of the application. Building the tests can be enabled using the script described in the previous section. Once the tests are built, they can be executed using the `ctest` command.

4.5.4 Deployment of Application

The application should be deployed in a format that is easy to use. This is achieved by providing an AppImage file for the Linux platform. For its creation, an ”AppDir” folder is included within the project. It contains the necessary files to create the AppImage, including the `.desktop` file and the icon used for the application. The icon was generated using the image generation features of ChatGPT. A script is provided to handle the creation of the AppImage. It creates a separate build folder where the application is built. For generating the AppImage file, the `linuxdeploy` tool is used along with a plugin for Qt applications.

For the deployment of the WebAssembly version of the application, the files generated by the build process need to be copied to a static web server. After that, the application can be accessed via a web browser. The files required for this are described in the project’s user guide.

4.5.5 Continuous Integration

To ensure that the application is buildable and that the tests pass correctly, a continuous integration (CI) system is used. For this purpose, a Jenkins server is employed. It uses the Docker container described in Chapter 4.5.1 to perform the builds. The server is private to the university and is not accessible from outside. It provides the capability to run the tests and build the application in different formats.

5 Reading and Parsing of Files

5.1 Reading of Yosys-JSON Files

The data from the Yosys JSON files needs to be read in. This is necessary to allow further processing of the data. This includes parsing into the internal data structures of the application, which were defined in Chapter 2.1.3. Reading files cannot be done by first getting the file path from the user using the `QFileDialog::getOpenFileName()` function and then opening the file to read its content [53]. This is due to the WebAssembly version of the application, where access to the file system is limited. It uses a sandbox environment that does not allow the application to read from the local file system without user interaction. This means that the functions that get the file path are only able to access the file system of the application inside the browser.

To get around this limitation, specific functions need to be used that use platform-specific APIs that allow access to the user's file system. For this, Qt provides the `QFileDialog::getOpenFileContent()` function, which does not return the file path but the content of the file after the user selects it [54].

5.2 Parsing of Yosys-JSON Netlists

The data needs to be converted into the internal data structures that describe a diagram in order to allow the application to route the circuit and to visualize it. This is done separately for each module that is contained within one netlist file. The steps needed to parse the data are discussed in the following sections. After the parsing of the data is done, the `Module` object is added to the `Diagram` instance. Additionally, if the module has the `top` attribute set as described in Chapter 2.1.3, it is set as the top module within the diagram.

5.2.1 Parsing of Objects

The data contained within the Yosys JSON file is read into the internal data structure. This is necessary before further processing can be done. The data is for the most part directly read into the internal data structure. The types parsed are the ones discussed in Chapter 2.1.3. The following shows the mapping of the object names inside the Yosys JSON file to the internal data structures.

- `Cell` → `Node`
- `Port` → `Port`
- `Netname` → `Netname`

When parsing `Netname` objects, the content of certain attributes needs to be changed to modify the content of the instance. For one, the `unused_bits` field contains the index position of bits within the netname's `bits` array that are not used. One example can be seen in Listing 4.4. These indices are read in, and the corresponding entries in the `bits` array are removed. Furthermore, if the `bits` array contains only the ”X” character, there is no way to match this name to any path object that gets created, so these `Netname` instances are dropped. The last thing that needs to be checked when parsing netnames is to avoid creating multiple instances of netnames with the same `bits` array. To ensure this does not happen, the final content of the array is checked against all previously parsed ones. If a match is found, no new `Netname` instance is created, but the name is added to the alias list in the existing `Netname` instance.

During the parsing of the `Cell` objects, additional `Port` objects are created that represent the pins of a cell. These are used to connect `Paths` to the cell's different pins.

5.2.2 Creation of Bus Splitters and Joiners

All the ports present in the circuit, whether they are part of a node or standalone ports, need to be connected. However, not every port matches another port directly. For example, a component may have an 8-bit connection where 4 bits are connected to an Adder and the other 4 bits to a Multiplexer. To visualize such connections in diagrams, bus splitters and joiners are used. These are special components that are not present in the netlist and therefore need to be created during the parsing of the diagram. For this purpose, an algorithm was created, which will be discussed in the following section. An example of a bus splitter and joiner, nodes, and standalone ports can be seen in Figure 5.1.

(a) Example for a node (FlipFlop) **(b)** Example for a standalone port (Input)

(c) Example for a bus splitter **(d)** Example for a bus joiner

Figure 5.1: Examples of Objects used in Schematics

5.2.2.1 Custom Algorithm

An algorithm was created to detect the need for bus splitters and joiners, which are special nodes. Additionally, the algorithm creates the `Path` object connections between nodes and standalone module ports. This algorithm uses `Path` objects that are created during the parsing of the netlist and does not require `Netname` objects. The `Path` objects are initialized without any information about their source or destination. The algorithm has two parts: the connection of the source to the `Path` instances, and the connection of destinations.

To connect the sources to the objects, all the sources of signals in the diagram, meaning ports of modules that have the ”input” direction and ports of nodes that are defined ”output”, are collected. These are then matched directly to the `Path` instances bit fields; if they match, the `Port` object is linked to it as the source of the `Path`’s signal. If they do not, it is checked whether the bits of the `Port` have a partial match to the signal. This is important when the `Path` object contains bits with the ”X” character. If this also does not succeed, a new `Path` object is created that contains the `Port` object’s bits as its own.

After the connection of the sources is completed, the destinations are connected to the `Path` instances. For this to work, all the destinations that are present in the module are collected. This includes all the module ports that have the ”output” direction and all the ports of nodes that are defined as ”input”. Then a seven-step algorithm is used to connect the `Path` instances to the destinations. In the following Table 5.1, the steps are described in detail. There, the words ”constant” and ”non-constant” are used when referring to the bits of a `Port` or `Path` object. A ”constant” bit value means that this bit does not connect to another port in the design but describes a connection to either ground or VCC, meaning the logic level of the bit is always 0 or 1. A ”non-constant” bit contains a pin number, meaning that if it has, for example, the value 2, it connects to all the `Port` objects that also have the value 2 contained in their bit array.

Step	**Description**	**Example**
1	Split a port’s bits into constant and non-constant bits. This is done because constant values need to be continuous, and each new one depicts a new value.	`Port[2, 3, ’1’, 4]` → `SubPort1[2, 3]`, `SubPort2[’1’]`, `SubPort3[4]`
2	Go through all the non-constant bit blocks and search for connections to existing `Path` instances. 1. Check if it matches the bits of a `Path` instance completely. 2. Check if it partially matches the bits of a `Path` instance. If it does, create it as a new instance and remember which `Path` it is part of.	1. `SubPort1[2, 3]` → `Path1[2, 3]` → `Path1[2, 3]` 2. `SubPort1[2, 3]` → `Path1[2, ’1’, 3]` → `SubPath1[2]`, `SubPath2[3]`
3	Check if the `Port` instance being checked was not split and only one `Path` instance was found. If this is the case, connect them and abort the algorithm.	
4	If not, check if a `Path` instance exists that contains the same bits as the `Port` instance. This is used to connect the `Port` to the created bus splitters and joiners. If it does not exist, create a new one.	`Path1[2, 3]`

5	Create a new standalone `Port` instance for each constant bit block, and additionally create a new `Path` object connecting to the `Port` instance.	`ConstPort[’0’, ’1’, ’1’]`
		→
		`ConstPath[’0’, ’1’, ’1’]`

6	Connect the `Path` instance found or created in Step 4 to the `Port` instance that is handled by the algorithm.	`Port1[2, 3]` → `Path1[2, 3]`
7	Creation of bus splitters and joiners. There are multiple cases for the creation of splitter and joiner instances: 1. The sub-`Path` instances created in Step 2 combined match the bits of the `Port` instance completely. ⇒ Create a joiner. 2. One sub-`Path` instance was created in Step 2, but the `Port` instance matches only parts of it. ⇒ Create a bus splitter. 3. If the first two cases apply in combination. ⇒ Create bus splitters and joiners.	1. `[2]`, `[3]`, `[2,3]` → Join(`[2], [3]`|`[2, 3]`) 2. `[2]`, `[2,3]` → Split(`[2,3]`|`[2]`) 3. `[2,4]`, `[3,5]`, `[2,3]` → Split(`[2,4]`|`[2]`), Split(`[3,5]`|`[3]`), Join(`[2], [3]`|`[2,3]`)

Table 5.1: Steps of the Custom Algorithm to Connect the objects in the Diagram.

The implementation of the algorithm has its limitations. Workarounds were needed to fix problems with missing connections on bus splitters and joiners or ports of `Nodes`. Such problems are, for example:

- A `Port` instance is only connected to a completely matching `Path` and not to a partial one, even when both are part of the bit vector.
- Creation of additional bus splitters and joiners even if they already exist.
- Multiple connections on bus splitters of the same signal, even though only one is needed.

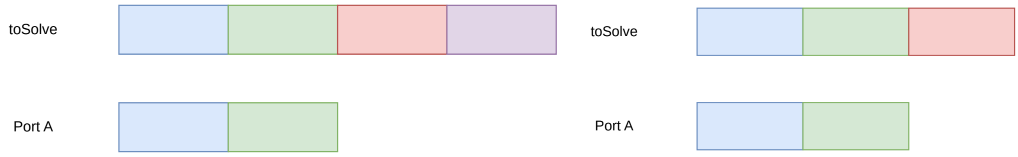
All these problems led to the decision to not develop the algorithm any further, but to search for already existing alternatives.

5.2.2.2 Netlistsvg Algorithm

To work around the limitations of the custom algorithm and the difficulties in fixing the problems with it, different connection algorithms were explored. The one contained within netlistsvg was adapted to work with the application. This was necessary because netlistsvg is written in JavaScript, and some built-in functions work differently or do not exist.

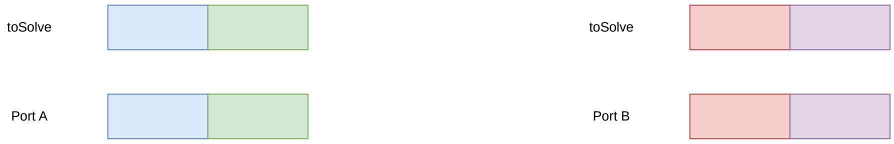
In order for this algorithm to work, all the standalone ports and ports of modules that serve as a destination for signals need to be modified. If they have constant bits in their bit vector, meaning bits that either have the value "0" or "1", they need to get a non-constant value. For this, all the ports are gathered, the constant parts of the bit vectors isolated, and then new bit numbers are sequentially assigned. The numbering starts after the highest existing non-constant bit number that is present in the module. During this process, a mapping is created between the vectors that contain the original constant values and the new ones with the non-constant values. This is done to allow the naming of paths/wires during the creation of `Path` objects.

The algorithm in netlistsvg is recursive. It takes all the source and destination `Port` instances and checks each destination `Port` object for needed bus splitters and joiners. It detects the need for a bus joiner by testing if the destination `Port` instance matches the bit vector of any source `Port` instance. If it does not, it removes the most significant bit of the vector to compare it against the source `Port` objects. It does this until it finds a match or the bit vector has been shrunk so much that it is smaller than the bit vector it is compared against. This can be seen in the Figures 5.2 and 5.3. Figure 5.3b shows that when a part of the bit vector is solved, the algorithm continues with the next unmatched part. In this case, a match is directly found.



(a) Step 1 (b) Step 2

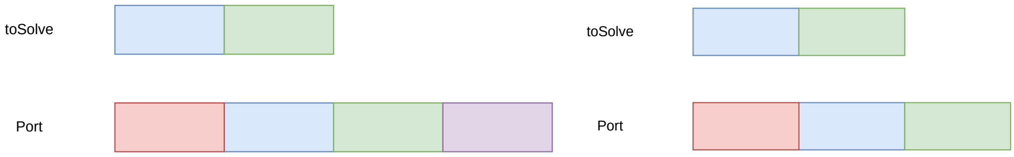
Figure 5.2: Creation of a bus joiner with netlistsvg Algorithm 1



(a) Step 3 **(b)** Step 4

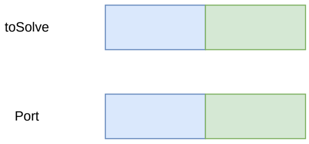
Figure 5.3: Creation of a bus joiner with netlistsvg Algorithm 2

The creation of splitters works similarly to the joiners. A splitter needs to be created when, after the reduction of the destination bit vector, only a part matches with the bit vector of a source `Port` instance. This can be seen in the Figures 5.4 and 5.5. If this is the case, an additional bit vector with the bits needed for this connection is added to the list of source `Port` instance bit vectors.



(a) Step 1 **(b)** Step 2

Figure 5.4: Creation of a bus splitter with netlistsvg Algorithm 1



(a) Step 3

Figure 5.5: Creation of a bus splitter with netlistsvg Algorithm 2

A combination of these two checks runs for every destination bit vector. Every time a part of the bit vector matches and there are still unmatched bits left, the algorithm continues to check for matches. To ensure that the algorithm can create all the bus joiners, the list of destination bit vectors is also checked for partial matches.

A bug was found during the implementation of the algorithm that caused connections to bus joiners to be missing when they were created out of a partial source bit vector match, when two different joiners used the signal. This was fixed by running the algorithm again over the current destination bit vector that is checked.

Due to the use of recursion by the algorithm, it was not possible to compute the bus splitters or joiners for larger diagrams when it ran in WebAssembly. This is caused by the limited stack size of the WebAssembly environment. To work around this problem, the algorithm was changed to use the `std::stack` data structure instead of recursion to store the data needed for each iteration.

5.2.3 Connection of Objects

The objects contained within the module being parsed need to be connected using `Path` objects. This is done to complete the model of the schematic in the internal data structures. The process is divided into two steps.

The first step is the creation of `Path` objects and the connection of signal sources to them. For this, all `Port` objects that serve as signal sources are gathered. Then, it is checked whether the bit vector of the `Port` instance matches those in a `Netname` object. If this is the case, it means that the netname can be used as the label or name of the `Path` object being created. Additionally, if the port is marked as constant, the translation mapping created during the conversion of non-constant bits to constant bits is checked. This ensures that the `Path` object being created also receives the correct netname if such a conversion took place. If no name can be determined by either of these checks, a name is generated from the name of the port followed by the "_sig" postfix. The name is then also hidden by default to ensure the readability of the schematic. With the determined name, the `Path` instance is created and connected to the `Port` object.

The second step is the connection of signal or wire destinations. For this, all `Port` objects configured as signal destinations are gathered. These are compared against the bit vectors of the `Path` objects. If the vectors match, the `Path` object is connected to the `Port` instance.

5.3 Symbols

5.3.1 Reading in symbols

The symbols contained in SVG files need to be read in to be used for the visualization of the circuit. They are provided in separate files, as this allows the user to load custom symbol files and adapt the schematic to their needs. This works similarly to the reading of the Yosys JSON files, which is discussed in Chapter 5.1. Additionally, the Qt resource system is used to load files that are embedded into the application, providing the program with default symbols. For this, the SVG file must be registered in the Qt resource system. It can then be loaded using the `QFile::open()` function.

To indicate that the file is a resource, the path to the file must be prefixed with a colon [57]. Further information can be found in Chapter 7.5.1.

The default symbols used are those provided by the TerosHDL project, which is licensed under GPL3 [43]. They have been modified to match the changes made to the netlistsvg format mentioned in Chapter 4.3.

5.3.2 Parsing symbols

The data contained in the SVG files needs to be converted into the internal data structures of the application. This allows the application to assign it to the different components in the diagram and to use it to visualize the schematic of the circuit.

In order to parse the read-in SVG data into individual symbol objects, the metadata needs to be extracted from the file. This consists of all the tags contained within the `<svg>` tag that are not group tags. Part of this is the `<svg>` object itself, which indicates that the data is in SVG format and defines the size of the image. Additionally, it contains formatting information for all the included elements. This includes, for example, line width and color. The metadata needs to be read in first, as it is used to create valid SVG objects for all the individual components that are part of the symbol file.

To create the `Symbol` instances from the data, each group tag that has the svg tag as a parent is evaluated. All the necessary information required for the creation of a `Symbol` object is extracted from the group tag, such as the name of the symbol and the bounding box dimensions. Additionally, each group within the symbols is evaluated to create the `Port` objects that are part of the symbol. In order to draw the SVG, the data also needs to be contained within the objects. For this, the group tag that defines the symbol is combined with the extracted metadata and stored inside the object as a string. This allows the application to use the SVG data for the visualization of the symbol. When creating the combined data, some attributes need to be adjusted to scale the SVG object to the correct size. These attributes are the `width` and `height` of the `<svg>` tag and the `transform` attribute of the group tag. The SVG size is set to match the bounding box of the symbol, and the `transform` attribute is set to the point $(0, 0)$ to position the symbol correctly during drawing.

6 Placing Components and Lines

6.1 Assignment of Symbols

The `Node` and standalone `Port` instances of the netlist need to be linked to `Symbol` objects. This is required for the routing algorithm to work correctly, because otherwise the algorithm is not able to calculate the positions of the components and the lines. The reason for this is the missing information about the size of the components and the position of the pins where paths/wires are connected.

To achieve this, the `type` of the `Node` or `Port` instances is compared against the name and the aliases of each `Symbol` object. If it matches, the `Symbol` is assigned to the `Node` or `Port` instance. This also allows the user to create custom symbols for specific modules that are present in a design. For them to be assigned, they need to have a name or alias that matches the type of the `Node` or `Port` instance. The same goes for the ports contained in the `Symbol` object. Further information about the creation of custom symbols can be found in the user guide of the project.

During the assignment of the `Symbol`, it is checked whether the `Node` is a variant that is used on a bus. For this, all the ports are checked for ports that have a width greater than one. Additionally, it is checked whether a bus variant of the type is available. If this is the case, the type is changed to the bus variant.

Furthermore, the types ”split” and ”bus” are handled differently. The same applies if the type cannot be found in the list of `Symbol` objects.

6.1.1 Creation of Bus and Generic Symbols

Symbols for bus connections and modules need to be created dynamically, depending on the number of ports. This is done because these symbols need to match the configuration of the `Node` instances. For example, if eight different signals need to be combined into a bus, a symbol is needed that has eight inputs and one output.

To generate these types of symbols, the existing template symbols present in the symbol file are used as a reference. The SVG data of these symbols is extracted and modified depending on the number of ports needed for the ”split”, ”join”, or generic symbol. The modified SVG data is used to create a new symbol, which is added to the list of symbols. The naming of the `Symbol` instance includes the number of inputs and outputs. This allows the program to generate new symbols only if they are not already present in the list of symbols, saving time and memory.

6.2 Calculation of Positions

The positions of components, meaning standalone module ports and nodes, need to be calculated in a 2D coordinate system. This is done to ensure a readable layout of the schematic, which the user of the application can understand. For this routing step, `libcola` is used. The reason for selecting this library is discussed in Chapter 3.1.4.

6.2.1 Converting to Cola Graph

The `Node`, `Port`, and `Path` instances need to be converted into cola graph objects. This is required for the cola library to be able to perform the routing of the objects in a 2D coordinate system.

6.2.1.1 Standalone Ports and Nodes

For the conversion of `Port` and `Node`, the information contained inside the `Symbol` instance assigned to them is used to generate the cola representation of the objects. This includes the size of the bounding box and the position of the pins.

To better understand this conversion process, a manual example is shown in Listing 6.1. The example shows how to create a module's input port where a signal from the outside is received.

Listing 6.1: Code for Input Port in libcola

```
// Symbol Rectangle
rect1 = new vpsc::Rectangle(0, 100, 0, 50);
rectangles.push_back(rect1);
// Port Rectangle
rect2 = new vpsc::Rectangle(100, 110, 20, 30);
rectangles.push_back(rect2);

// create edge
edges.push_back(std::make_pair(0, 1));

// create constraints
sep1 = new cola::SeparationConstraint(vpsc::XDIM, 0, 1, 55, true);
constraints.push_back(sep1);

sep2 = new cola::SeparationConstraint(vpsc::YDIM, 0, 1, 0, true);
constraints.push_back(sep1);
```

For the bounding box, a rectangle is created that matches the proportions of the box. For each port object contained in a `Symbol` instance, a port rectangle is created. These have a fixed size and are positioned at the coordinates that are defined in the data structure. These coordinates are relative to the bounding box and are added to the fixed size of the port rectangle to get the correct position. This can be seen with the rectangle creation in Listing 6.1. To ensure that the port rectangles do not move during the routing process, they need to be fixed in position. For this, each port first needs to be connected to the bounding box rectangle with an edge. This is done by creating a tuple inside the edge vector. The numbers of the rectangles that are connected correspond to their index in the vector containing the rectangles. For example, when rectangles 0 and 1 are to be connected, the tuple added to the edges would be $(0, 1)$. To fix the position of the port rectangles, they need to have constraints added to them for both the x and y directions. These are calculated so that the port rectangle's center lies at the position of the pin. These values are calculated from the center of the rectangle with the following formulas for the x and y directions:

For the x direction:

$$x_{const} = x_{port_{center}} - x_{bound_{center}}$$

For the y direction:

$$y_{const} = y_{port_{center}} - y_{bound_{center}}$$

For the example in Listing 6.1, where the port is at the position $(100, 0)$, the values are $x_{port} = 105 - 50 = 55$ and $y_{port} = 0 - 0 = 0$. The calculated values can also be negative, meaning that the port is to the left for x_{const} and to the bottom for y_{const}. These constraints are also saved inside a vector for use in the routing algorithm.

The result created by the manual example explained and shown in Listing 6.1 was exported using the debug functions of `libcola` and can be seen in Figure 6.1.

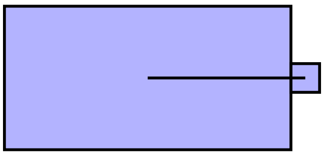


Figure 6.1: Example of a Input Port as a cola Graph Object

For better routing results, each generated cola representation is grouped into so-called ”clusters”. These are groups of rectangles and edges which can define a padding around them into which no other cluster can intrude. The parameters used for setting the padding are further explained in Chapter 6.2.2.

6.2.1.2 Paths

In order to create the connections between the standalone ports and nodes, the `Path` instances also need to be converted, because they contain the information about the connections between objects. This works similarly to the creation of fixed ports shown in Chapter 6.2.1.1. Each `Path` instance is converted by creating an entry in the edge vector between the rectangle that represents the source port and each of the rectangles that represent one of the destination ports. Different from the creation of the standalone ports and node representations, the constraints are set so that the connections can be routed more freely by the algorithm. For this, routing parameters need to be calculated. At the beginning constant values where chosen for the different routing parameters.

6.2.2 Calculation of Routing Parameters

The parameters used for routing the schematic need to be determined. This is done to ensure that the routing algorithm provides the best possible results for as many cases as possible. Therefore, experiments with different diagrams and parameters were conducted. The parameters are described in Table 6.1. These experiments were conducted with a finished version of the program, which is able to display the diagrams in a Qt GUI and includes settings to adjust the parameters of the algorithm.

Name	Description
x constraint	The x constraint is used to set the x distance for the edges that represent the `Path` objects.
y constraint	The y constraint is used to set the y distance for the edges that represent the `Path` object.
test precision	This is a value that determines when the routing algorithm's results are sufficient.
test iterations	The number of iterations the routing algorithm performs before giving up.
default edge length	The length of the edges before routing.
node edge length	The length of edges that connect two nodes together before routing.
port edge length	The length of edges that connect a standalone port to a node before routing.
cluster padding	The padding is the distance between each cola cluster. A cluster is a subset of cola rectangles and edges, and the padding forms a box around the cluster that cannot be crossed by other clusters.

Table 6.1: Routing Parameters

Before starting to calculate the parameters, they were set to large values for all netlists to ensure that the routing with the cola library produces readable results for a large number of cases. This presented a problem for small diagrams, where the user received a schematic in which the components are placed so far apart that it is hard to read. It could also happen that components are placed on top of each other.

This was solved by calculating the routing parameters according to the size of the netlist. For this, another experiment was conducted where the size of the netlists was recorded. Then, the routing parameters were adjusted until the diagram's routing results were readable, so that no blocks are placed into one another and the distance between the components is not too large. During this experiment, it was discovered that the edge lengths for nodes and paths do not need to be set to different values and were combined. Furthermore, the padding of the clusters was set to a constant value. It was also discovered that the test precision and test iterations have little to no effect on the routing results, so they are also kept at a constant value. The amount of ports is a metric that includes the ports of nodes and the standalone ports of the netlist.

The results of the experiment can be seen in Table 6.2.

Name	**#Ports**	**#Nodes**	**#Standalone Ports**	**Edge Length**	**X Const**	**Y Const**
MAdderCore	28	16	7	10	75	75
VISCY-V	14	7	5	10	75	75
MAdderCore Tech-(ECP5)	71	13	10	10	100	100
baudgen	49	10	13	10	75	75
byteselector	120	34	19	10	250	250
datahandler	67	11	18	10	125	125
alu	151	32	28	10	250	250

Table 6.2: Routing Parameters from Experiment

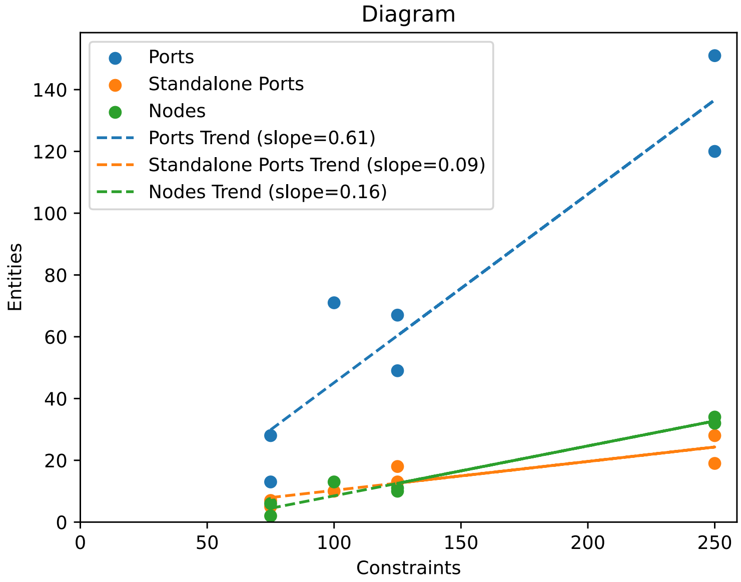


Figure 6.2: Graph of Experiment Results

Out of the results, a graph was created, which is depicted in Figure 6.2. This was done with a Python script that was created with the help of ChatGPT. This graph shows the number of ports, standalone ports, and nodes in comparison to the set x/y constraints. It was possible to create a graph that combines both constraints because they are set to the same value. Furthermore, no graph for the edge length needs to be created because it had no significant effect on the desired routing results. As can be seen from the trend lines in Figure 6.2, the increase in the different types of objects has a linear relationship with the increase of the constraints. In order to create a formula to calculate the rise in the constraint values with the size of the netlist, the slopes of all the trend lines need to be taken into account. This is required because, as can be seen in Table 6.2, none of the amounts on their own correlate with an increase in the constraints. For example, between the netlists ”MAdderCore” and ”baudgen”, the amount of ports doubled but the constraints stayed the same. To combine the values and to get the separate constraint values that are related to the amount of ports, standalone ports, and nodes, the following formula is needed:

$$C_x = \frac{m_x}{\#x}$$

With x being the quantity of the object the constraint is related to. To combine the values, an average of the three constraint values is calculated, which is the final constraint value used for the routing of the netlist. This, together with the other values that are kept constant, is used as parameters for the cola library. Even with these values, the routing algorithm sometimes produces results that are not readable.

This led to the decision to examine the programs mentioned on the Adaptagrams project website that use their libraries [6]. When searching through these programs, it was found that the program Inkscape uses `libcola`. Instead of using the `ConstrainedFDLayout` algorithm, Inkscape uses the `ConstrainedMajorizationLayout` algorithm. This algorithm is marked as deprecated in the cola library [2]. An attempt was made to adjust the code to use `ConstrainedMajorizationLayout` to see if it produces better results in comparison to the `ConstrainedFDLayout` algorithm. This was not the case, which is the reason why the project was reverted back to the original code.

6.2.3 Routing of Positions

The routing of the object's positions is done with the calculated parameters. For this, the algorithm is executed. It runs until the constraints are met or the maximum number of iterations is reached. The results of the routing algorithm are stored inside the instances of nodes and standalone ports. This data can be used for the further routing process, which is done with `libavoid`, and for the graphical display of the objects. Information on the runtime of the routing can be found in Chapter 8.1.

6.3 Calculation of Lines

The lines that are used to connect the standalone ports and nodes need to be calculated. This needs to be done because the connections made by `libcola` are not avoiding any obstacles. To ensure that the lines go around objects, `libavoid` is used.

6.3.1 Conversion to the `libavoid` format

The `libcola` representation of the schematic needs to be converted to `libavoid` objects. This is required because the avoidance routing algorithm implemented in the avoid library only works on its own data structures.

6.3.1.1 Conversion of Cola Rectangles

The rectangles present in the cola graph depict two things. For one, they represent the bounding boxes of symbols for the standalone ports and nodes. The other thing is the connectors/ports that each symbol has for the connection of wires/paths. Both need to be handled differently. The bounding box rectangles can simply be converted to `libavoid` rectangles that have the same position in 2D space. The connectors are detected by the specific size of the rectangles. When one is detected, the side the connector is placed on needs to be calculated. This is done by comparing the center point of both the rectangle that describes the bounding box of the symbol and the rectangle of the connector. Depending on this, the side is set, which is important for `libavoid` because this determines the direction the edge will be routed into the connector of the symbol. While converting connectors, it is also important to allow multiple connections to the same one to allow for the connection of multiple paths from one signal source to different destinations. The connectors are also mapped to the index of the cola rectangles inside the rectangle vector to allow the creation of edges in the avoid library.

6.3.1.2 Conversion of Edges

The edges are created by going through the edge vector of the `cola` graph and finding the IDs of the rectangles inside the mapping of `cola` rectangles to `libavoid` connectors. The two connectors are then connected together with a `libavoid` edge.

6.3.2 Configuration of `libavoid`

The router of `libavoid` needs to be configured. This is done to get the best possible results for the routing of the lines. For one, the type of routing needs to be set. The library supports two types of routing, ”orthogonal” and ”polyline”. For a schematic viewer, the use of ”orthogonal” is better suited. Additionally, by default, `libavoid` moves lines apart as soon as possible. This is not desired when creating a schematic viewer because it makes a diagram unreadable. It should keep the lines that are connected to the same connector together as one for as long as possible, to improve the readability. The last settings that need to be changed are the distance between two lines and the distance a line needs to have to a rectangle in the avoid graph.

6.3.3 Improved Performance of `libavoid`

The performance of the line routing algorithm needs to be high, meaning that the runtime of the algorithm should be as low as possible. This ensures that the user has a good experience when using the program and does not have to wait a long time for the diagram to be displayed. When testing the finished program, it was discovered that the routing times of `libavoid` are very high for more complex diagrams. More information about the runtimes can be found in Chapter 8.1. To identify the cause of the high runtimes, the program was run through a profiler. With its help, it was possible to find the functions that take the most time. These are the functions `canFollowLeft()` and `canFollowRight()`. Their code is shown in Listing 6.2.

Listing 6.2: Unmodified canFollow Functions

```
inline bool Block::canFollowLeft(Constraint const* c, Variable const*
     last) const
{
   return c->left->block == this && c->active && last != c->left;
}
inline bool Block::canFollowRight(Constraint const* c, Variable const
    * last) const
{
   return c->right->block == this && c->active && last != c->right;
}
```

As can be seen in the listing, the functions only check a few conditions. During this check, the function dereferences the `left` or `right` member of the constraint `c` twice, which, when called frequently, can lead to increased runtimes. To mitigate this, the code was changed as shown in Listing 6.3, by first retrieving the content of `left` or `right` and then checking the conditions, reducing the number of dereferences to one.

Listing 6.3: Modified canFollow Functions

```
inline bool Block::canFollowLeft(Constraint const* c, Variable const*
     last) const
{
   const auto* left = c->left;
   return left->block == this & c->active && last != left;
}
inline bool Block::canFollowRight(Constraint const* c, Variable const
    * last) const
{
   const auto* right = c->right;
   return right->block == this && c->active && last != right;
}
```

To further improve the performance of the algorithm, a condition was added to check if the `active` member of the constraint instance is set before each call to the `canFollowLeft()` and `canFollowRight()` functions. If this condition is not met, the function call is skipped, further saving time. The complete code change can be seen in the appendix in Listing A.1.

The changes were tested with multiple diagrams. The results can be seen in Chapter 8.1.

6.3.4 Routing of Lines

The routing of the lines calculates the position of bends in the lines. The path that results from it is then stored inside the `Path` instance to allow the drawing of them in the GUI. This is done by running the routing algorithm of `libavoid` on the graph that was created in the previous step. The routing algorithm runs until it determines that the constraints of the graph are met. The results are stored within the `Path` instances so that they can be used for the creation of the graphical representation. Information about the runtime can be found in Chapter 8.1.

7 Visualization and User Interface

7.1 Overview over Graphical User Interface (GUI) Features

The graphical user interface (GUI) of the application needs to include functions that allow the user to interact with and change settings of the application. This is necessary so that the user can view diagrams and adjust the view to their needs. In the following, the features needed to accomplish this goal are described:

- The application needs to display the names of the signals, standalone ports, and submodules. Additionally, the values of constant ports also need to be shown. This allows the user to easily identify which parts of the schematic they are viewing, improving the readability of the schematic.
- Symbols need to be rendered at points where paths with a common source diverge. This is done to show the user that the paths are not crossing but splitting.

- The user needs to be able to navigate through the submodules of the design. This is required to provide a seamless experience when viewing the schematic, so that the user does not have to open a new instance of the application to view another module. For this, it is also important to give the user feedback about the hierarchy of the design, so they know which module they are currently viewing.
- The application needs basic navigation functionality within the currently viewed module. This allows the user to view the parts of the schematic they are interested in. For this, features such as zooming, panning, and selection are needed. These features should also have shortcuts so they can be accessed quickly.
- Additional functionality for modifying the currently viewed schematic is needed to allow the user to see more information about the schematic or to locate specific constructs, such as combinatorial loops, in the diagram. For this, functions like module search, object highlighting, and a window to display additional information are required. This may also include features that allow the selection or highlighting of all neighboring objects, or functionality to zoom in on specific objects.
- The user should have access to settings for the application. These allow the user to modify the appearance of the schematic. For this, it should be possible to change the symbols used in the diagram. Furthermore, the routing parameters should be adjustable to allow the user to optimize the routing when the default values do not produce the desired result.
- The user should be able to export the schematic to an SVG file. This allows them to use the schematic in other programs. Exporting to the SVG format also enables scaling of the image without loss of quality, which a screenshot would not allow. It should also be possible to export only parts of the schematic.
- The application should include example diagrams. This allows the user to test the program without having to install the OSS CAD Suite or Yosys. It should include different examples to demonstrate all the features of the application.

7.2 Converting to Qt Objects

The objects created by the routing libraries need to be converted into Qt objects. This is done so that the routed schematic can be displayed in the GUI of the application. To achieve this, the three different types of objects, `Path`, `Node`, and `Port`, with `Port` referring to the standalone ports used in a schematic to display connections to and from the module, need to be converted separately into Qt objects. This allows easier access to the information stored within the objects, which includes the used symbols, the coordinates for objects like nodes and standalone ports, or the points that form the line of a path.

7.2.1 Nodes and Standalone Ports

The nodes and standalone ports need to be converted to Qt objects that can render the SVG representation of the assigned symbol. This is done to create a visible diagram from the internal representation of the schematic. SVG symbols are used to allow the use of custom symbols. To convert the objects into ones usable for creating a visual display in Qt, only the coordinates, measured from the top left corner, and the SVG representation of the symbol are needed. Additionally, a reference to the object is stored to allow access to the information stored in the diagram object. The coordinates are stored in the `libavoid` representation of the object, which stores the center of the object. To obtain the coordinates of the top left corner, the proportions of the bounding box stored in the associated symbol are used. The SVG representation is added by using a renderer that is shared across all symbols of the same type, saving memory, since the SVG renderer is only created once per symbol type.

7.2.2 Paths

The paths need to be converted to Qt objects that can be rendered in Qt. This is required to create a visible diagram from the internal representation of the schematic. To convert the data contained inside the `Path` objects, each subpath needs to be merged into a single Qt object. Subpaths represent the connection from the signal's source to one of the destinations of the signal. If there are multiple destinations, the subpaths are merged into one Qt object. For this, the points created by `libavoid` are used. Each subpath is added point by point into the Qt object. When another subpath is added, the positions where the subpaths diverge from one another need to be marked with a dot to show the user that the signals are not just crossing, but that the two paths are splitting.

In Figure 7.1, an example of two subpaths is shown that both start at the source S and thus need to have a dot at the diverging point. The colored dots mark the points that were generated by `libavoid`. When combining both paths together, as shown in the right path in Figure 7.1, it is possible to see which points overlap by the combined color. To find the point where the subpaths diverge, the points

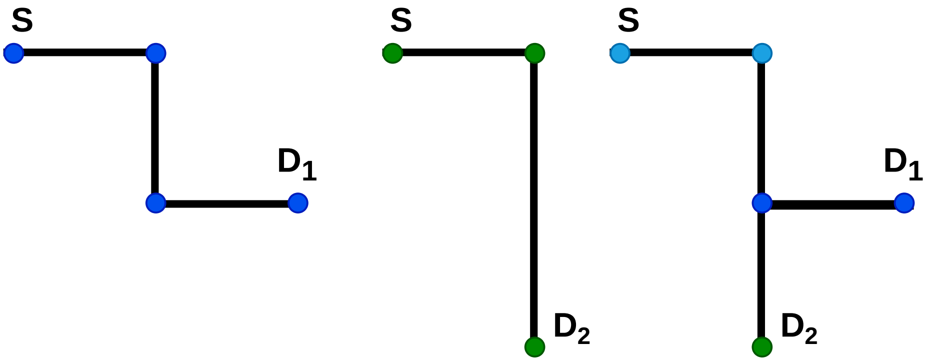


Figure 7.1: Example of Points Generated by the Routing Library

are compared one by one until they no longer match. To find out which diverging point needs to be marked, a comparison is made. It is checked which one of the diverging points is closer to the last matching one. The point that is closer still lies on the other path and thus is the one that needs to be marked. This method works because it is guaranteed that the paths generated by the avoid library are orthogonal, meaning they only have straight lines and 90-degree bends. Therefore, another point is needed to change the direction. The result of this process can look like Figure 7.2 when displayed in the Qt GUI.

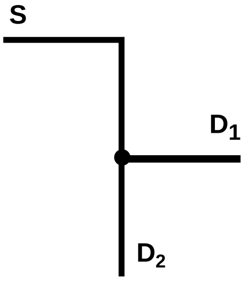


Figure 7.2: Example of a Path with Diverging Point

The process is then repeated for all the subpaths until all of them are added to the Qt object. Additionally, a reference to the diagram's `Path` object is added to easily access the information stored in the diagram object.

7.3 Rendering Names

The names of signals, standalone ports, and submodules need to be rendered in the GUI. This is done to add more information to the schematic, which makes it easier to identify sections of the schematic. The names are rendered as text items in the Qt scene and are not part of the SVG symbols used, as done in netlistsvg. This decision was made because it would require a separate renderer for each symbol that contains text. There are a few different types of objects that need to be named. These include module blocks that represent submodules, standalone ports as connectors to external signals, and paths.

When naming the paths that are drawn between the nodes and standalone ports, the name is rendered only when the ”hidden” flag was not set in the netlist or during the creation of the `Path` object. If it is not hidden, the name is rendered at all ends of the path. If the signal is wider than one bit, the width of the path is also added. When an end of the path is connected to a bus splitter or joiner, only the position of the bits inside of the whole signal is displayed. This means that if the signal receives the top two bits at a bus splitter from an 8-bit signal, the displayed name would be ”<7:6>” instead of the name of the signal itself.

For standalone ports, there are two different types of names. The first is for inputs and outputs, which are labeled with the name of the port. For constant values, which are also handled like standalone ports, the value of the constant is displayed.

When labeling nodes that depict submodules, the name of the submodule is displayed along with the name of the specific module instance. For example, when the module is called ”adder” and the instance is ”i0”, the name shown would be ”adder:i0”. Additionally, the signals that are connected to the modules are not labeled with their own name but with the names of the ports that are present. This is done to make it easier to identify which signal is connected to which port.

7.4 Visualizing the Schematic

The Qt objects need to be visualized in the GUI. This is required to display the schematic to the user. For this, the Qt objects are added to a scene, which can be displayed inside a view widget provided by Qt. Both the scene and the view have custom implementations to allow the addition of custom features needed for the application. The display of a simple adder in the Qt GUI window can be seen in Figure 7.3. When a netlist file is selected to be displayed, the diagram always displays the top module of the design.

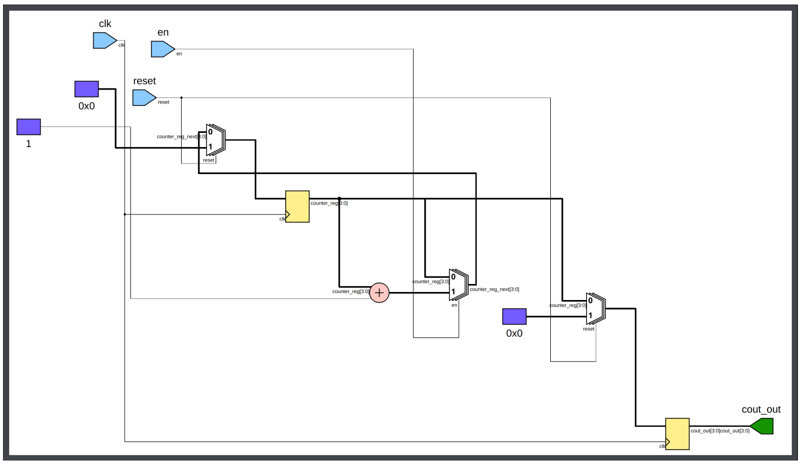
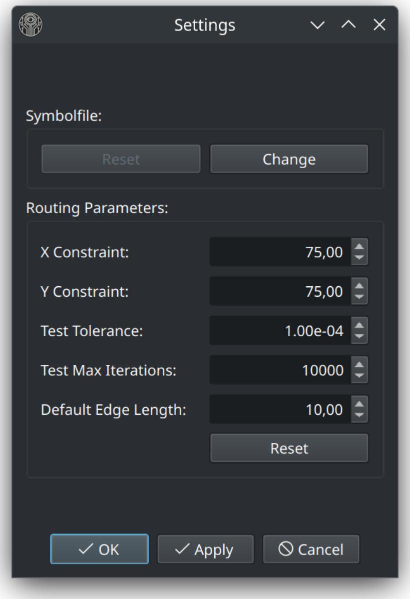


Figure 7.3: Example of a Schematic Displayed in the Qt GUI

7.5 Settings


Settings

Symbolfile:

Routing Parameters:

X Constraint: 75,00

Y Constraint: 75,00

Test Tolerance: 1.00e-04

Test Max Iterations: 10000

Default Edge Length: 10,00

Figure 7.4: Settings Dialog

7.5.1 Settings for Symbols

The symbols need to be customized to allow the user to replace them with custom symbols or add some for specific modules. For this, the ”Symbolfile” in the settings, which is depicted in Figure 7.4, is used. This allows the user to load a custom symbol file. For the symbols to be loaded correctly, it is required that they adhere to the

Symbol format. More information about it can be found in Chapter 4.3.1. After loading a custom set of symbols, all the routing results are invalidated because the change in size requires the positions and lines to be recalculated. There is also the option to reset the symbols to the default ones. This is done by loading them from the Qt resource system because they are embedded into the application. Further information can be found in the user guide of the program.

7.5.2 Settings for Routing

The routing parameters need to be changeable by the user. This is done to allow adjustments to the routing if the calculated parameters produce an undesired result. As can be seen in Figure 7.4, the parameters that were discussed in Chapter 6.2.2 are shown. These parameters depend on the module that is displayed. If the module is changed, the parameters shown in the settings dialog are also updated. This allows adjustments to specific modules without affecting others. When the parameters are changed by the user, the routing is invalidated, meaning that the module needs to be routed again. This also works when a submodule of the schematic is selected. Further information can be found in the user guide of the program.

7.6 Navigation for Submodules

The navigation of modules and submodules needs to be possible. This allows the users to view not only the top module but also the submodules without having to open a new instance of the application or generate a netlist that only contains the submodules. For this, two different ways of navigation are implemented in the application. The first way is to do this directly in the schematic being displayed by double-clicking on a module. The other possibility is to use the hierarchy browser integrated into the application, which is shown in Figure 7.5.

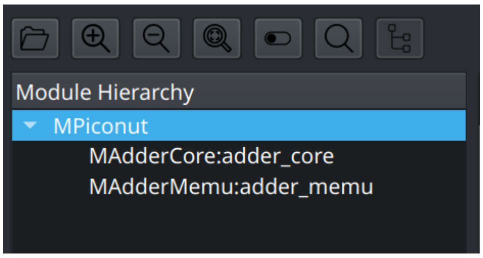
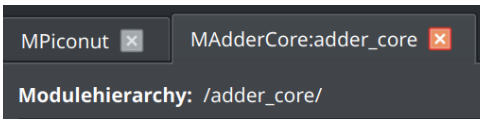


Figure 7.5: Hierarchy Browser

It displays the complete hierarchy of the design as a tree. The entries are named after the module name followed by the instance name. By clicking on one of the entries, the selected module is displayed in the schematic view. The hierarchy browser has the advantage that the user can see all the modules at once and does not have to navigate through multiple levels of submodules before finding the desired one.

Both ways of navigation trigger the creation of a tab in the GUI that shows the schematic of the selected module. This is depicted in Figure 7.6.



Modulehierarchy: /adder_core/

Figure 7.6: Tabs for Submodules

On submodules, the "Modulehierarchy" is also displayed. This functions like a file path to show the user where the current tab's module is located in the hierarchy. Additionally, the corresponding entry in the hierarchy browser is marked. Both of these features make the navigation through the design easier by giving feedback to the user about where they are in the hierarchy. Further information can be found in the user guide of the program.

During the development of the application, a problem was discovered where the application had a probability of hanging when navigating through modules. However, it only occurred when the AppImage or ELF file was run natively on a Linux system. The problem did not arise when it was run from within the Docker container or as the WebAssembly version. Through the use of `strace`, it was discovered that the program hangs on calls to `brk()`, which sets the end of the data segment of a process [32]. This is used by `malloc` to allocate memory [33]. Research with ChatGPT suggested that replacing the standard `malloc` implementation, provided by glibc, with `jemalloc` could solve this problem. This implementation of `malloc` uses `mmap()` to allocate memory, which can prevent race conditions that might have been the cause of the issue. This is also described as a benefit of the `jemalloc` library in its man page [27]. This was tested using `LD_PRELOAD` and resolved the issue of the application hanging. For the final implementation, the `jemalloc` library was added to the requirements of the native build, the build system, and the Docker container.

7.7 Functionality of the User Interface

7.7.1 Loading of Netlists

The application needs to be able to load netlists. This is required to generate a graphical representation of the netlist. When loading a diagram, the application can generate a warning message that informs the user that the netlist has surpassed a certain size, as the routing can take significantly longer. Information about the routing times can be found in Chapter 8.1. No progress bar was implemented because this would have required the use of threads to run the routing and GUI elements simultaneously. This needs a separate version of the WebAssembly Qt library, which might introduce additional bugs. This is the reason why it was not implemented in this thesis.

Another message box is displayed when a second netlist is loaded to inform the user that the current schematics will be closed. This is done to prevent the user from accidentally replacing the current schematic with a new one. Further information can be found in the user guide of the program.

7.7.2 Zooming and Panning

The schematic needs to be zoomable and pannable. This is done to allow the users to set the focus of the view on the area they are interested in. For this, standard Qt functionalities are adjusted to provide the ability needed for this program, using common shortcuts like ”Ctrl + Mouse Wheel” to zoom in and out. The implemented ”Zoom to Fit” function is also automatically triggered when a new tab for a module is created to ensure that the schematic uses all the available space in the view. Additionally, it is possible to zoom in on a specific symbol with the use of the context menu that opens when right-clicking on a symbol. This also works when searching for the name of a symbol, signal, or standalone port through the search menu. Further information can be found in the user guide of the program.

7.7.3 Highlighting/Selection of Objects

The objects in the schematic should be selectable and highlightable. This enables the user to easily trace signals through a schematic, which might help when searching for combinatorial loops. In the application, selection means that one or multiple objects are marked by clicking or dragging a rectangle around them. This selection is removed when clicking on an empty space or another object without using the ”Ctrl” modifier. Highlighting, on the other hand, is done by using the context menu and choosing a color. This marking is not removed by clicking on other objects or empty space, but only by using the context menu or the clear button, which removes all highlighting. To make it easier to add selection or highlighting to trace the signal path through the schematic, functions are provided that, for example, add a selection to all the destinations of the path that was right-clicked. Further information can be found in the user guide of the program.

In the beginning, there was a problem with the selection of paths. When looking at Figure 7.7, there are two marked points. When the user clicks on these positions, it is expected that the path becomes selected, meaning it gets colored in red as can be seen. However, without adjustments, the selection would not mark this path but the one labeled ”reset”. This was caused by the shape that Qt uses to determine if a click registered on a specific graphics object. This could be fixed by adjusting the generation of the shape to only include the space that the path occupies.

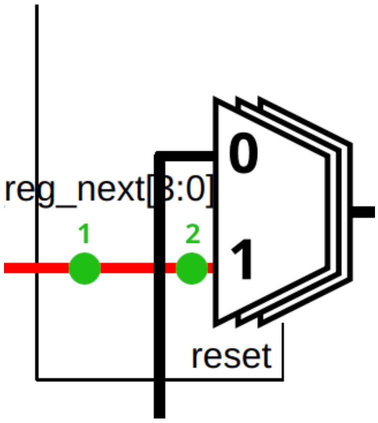
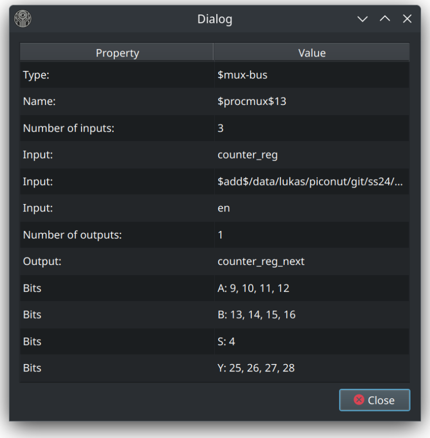


Figure 7.7: Problem with Selection of Paths

7.7.4 Display of Additional Information

The application needs to be able to display additional information about all objects that are present in the schematic. This allows the user to better understand the schematic and the connections within it without overloading the graphical representation. The information can be shown by clicking on ”Properties...” in the context menu of any object. This opens a dialog that looks like the one shown in Figure 7.8.

This dialog gets filled with information depending on the object that was clicked on. For example, when clicking on a path, the names of the source and destination ports are displayed, which do not exist on nodes or standalone ports. It also displays the names of signals that have the hidden flag set in the netlist, which allows the user to trace signals even if they are not added to the schematic and would only overload it. For example, if the name is a path on the user’s system. Another example is the type of nodes, which, for instance, could be ”$add”, indicating that the node is an adder. Further information can be found in the user guide of the program.



Property	Value
Type:	$mux-bus
Name:	$procmux$13
Number of inputs:	3
Input:	counter_reg
Input:	add/data/lukas/piconut/git/ss24/...
Input:	en
Number of outputs:	1
Output:	counter_reg_next
Bits	A: 9, 10, 11, 12
Bits	B: 13, 14, 15, 16
Bits	S: 4
Bits	Y: 25, 26, 27, 28

Figure 7.8: Properties Dialog Example

7.7.5 Loading Examples

The application needs to be able to load example netlists that are provided with the program. This is done so a user can test out the application without having Yosys or the OSS CAD Suite installed, to see if the application is useful for them. This is done similarly to the default symbols, meaning that the example netlists are embedded into the application. They include examples that show the display of simple netlists, designs with submodules, and technology-specific netlists like the ECP5 FPGA. Further information can be found in the user guide of the program.

7.7.6 Export of Schematics

The application needs to be able to export the schematic. This allows the user to create graphics of the schematic that they can use in documentation without having to take a screenshot. This is an advantage because the export is done in the SVG format, which allows the user to scale the image without losing quality. The program allows two different types of exports. The first one is an image of the complete schematic. The second one only exports the currently selected objects. For the SVG export, the Qt SVG library is used. In order for this to work correctly and use the SVG representation of the symbols instead of a generated bitmap, it was necessary to disable caching on the SVG symbol objects within the Qt representation of the schematic. Further information can be found in the user guide of the program.

7.8 Command Line Arguments

The application needs to be able to be used from the command line. This allows the tool to be used inside scripts, so the display of the schematic can be, for example, integrated into the build system of an HDL project. The command line argument parser is implemented using the Qt library. It allows users to load netlists directly from the command line and additionally change the symbol file used to generate the schematic that is being displayed. Information about the parameters that are available can be found in the user guide of the program.

7.9 Peer Test of the Application

The application needs to be usable for users who are not familiar with it. To ensure this, a peer test was conducted by members of the EES group to provide the author of this thesis with feedback about the usability of the application and to identify features that are missing or need to be adjusted.

During the peer test, the testers were asked to read the user documentation, build the program using the provided instructions, run the application, and test it with different netlists of their choice.

Table 7.1 shows the feedback given by the testers and what has been done to incorporate it into the program.

Feedback	**Action**
Multiple testers reported that netlists missing the `top` module would simply open a blank schematic view when Yosys synthesis was not run with the `hierarchy -auto-top` option.	The `top` value is the only way to detect the top module of a design, so it must be present in the netlist. A warning was added to the application to inform the user that the netlist does not contain a module with the `top` attribute, and that synthesis with Yosys using `hierarchy -auto-top` is required.
When opening a netlist using the command line, the resulting diagram was not fitted to the view, so the diagram appeared too zoomed out for the user to see.	The zoom-to-fit function is now called after the GUI is fully loaded and visible.

Multiple testers reported that diagrams created with the program are too spread out, making it hard to identify components of the schematic when the whole diagram is displayed.	This was improved by calculating routing parameters based on the number of objects in the schematic, as described in Chapter 6.2.2. Though it is not perfect and still leaves some empty spaces, it provides a more readable layout for different design sizes. The focus was on preventing overlapping nodes or lines, as overlap hinders readability more than empty space. Since preventing both overlap and empty space is not possible with the Adaptagrams libraries, overlap prevention was prioritized.
The application should have a progress bar when loading diagrams so that the user knows the application is not frozen.	This was not implemented because it would require the use of threads, as described in Chapter 7.7.1.
Multiple testers suggested adding the name of the file or design to the title bar or footer of the application to make it easier to identify instances of the application.	The name of the file is now displayed in the title bar of the application after it is loaded.
It was suggested that example diagrams should be included in the application so that the user can test it without having to install the OSS CAD Suite or Yosys.	Example diagrams are now included in the application. They are embedded into the application and accessible from the ”File” menu. This is further described in Chapter 7.7.5.
It was suggested that some option texts should be adjusted to make them more readable. For example, in the ”Export” menu, the file type should be added to the export options.	This was done as the testers suggested.
It was noted that the runtimes of the routing are rather high when using medium to large designs.	This is correct. The cause is the path routing algorithm in `libavoid`. Though it was improved, it is still too slow for fast routing of larger designs. More information about the runtimes can be found in Chapter 8.1.

Suggestions for improvements to the user guide were made by multiple testers.	The feedback was incorporated into the user guide.

Table 7.1: Feedback from Peer Test

8 Results

8.1 Runtime

This chapter presents the results of the runtime analysis of the application. Three different values were measured: the time it takes to perform the `cola` layout, the time it takes to perform the `avoid` layout, and the time the whole application takes from the assignment of the symbols to the finished graphical display.

8.1.1 Setup

The setup of the runtime analysis was as follows:

- Computer:
    - CPU: AMD Ryzen 7 5800X
    - RAM: 32 GB
    - GPU: NVIDIA GeForce RTX 3070
- Operating System: Debian 12.10 (April 2025)
- Browser (WASM): Google Chrome 134.0.6998.88
- Qt Version: 6.8.1
- emscripten version: 3.1.56
- Built in docker container

- Run in docker for native build
- Run on host for WASM build (browser)

The application was built using the default `setup.sh` script and the `$ ninja` command. The application was modified to add time measurements to the `cola` and `avoid` layouts as well as to the whole display process. This was done using C++ standard library functions from the `chrono` library. The results were printed to the console.

Three different test cases were run:

- unmodified `libavoid`
- modified `libavoid`
- modified `libavoid` in WebAssembly

The modification was made during the development of the application to improve the performance of the `avoid` layout. More information about the changes can be found in Chapter 6.3.3.

The last test was performed using the WebAssembly build of the application to compare its performance with the native build. For this, the build was configured with the `setup.sh` script, as with the native builds. The only difference was that the native Qt library was replaced with the WASM version. For this, the `QT_DIR` environment variable, which points to the library used by `setup.sh`, was changed to the install location of the WebAssembly version in the Docker container. This test also uses the improved `libavoid` library.

For all test cases, the application was run five times. The same five designs were used in each case. For all of them, the routing parameters for `libcola` were calculated and set as described in Chapter 6.2.2. The settings for `libavoid` were also set as described in Chapter 6.3.2. The value for the distance between paths is set to 7.5, and the distance between nodes, standalone ports, and paths is set to 10.0. These values are constants in the program and cannot be set by the user. The only value that is calculated for each design is the X/Y constraint. The following sections contain all the indicators and a short description of the designs used for the tests.

8.1.1.1 Design: `regfile`

This design is a simple register file with 32 registers, each with a width of 4 bits. It has one selectable input and two selectable outputs. The indicators of the design are shown in Table 8.1. A screenshot can be seen in Figure A.1.

#Nodes	#Ports	#Standalone Ports	#Paths	Computed X/Y Constraint
31	201	23	54	259.60

Table 8.1: Design `regfile` Indicators

8.1.1.2 Design: `VISCY-Controller`

This design is the controller of the Very Reduced Instruction Set Computer (VISCY) processor. It is a simple 16-bit processor with a custom instruction set developed by the EES group for use in education [29]. This is a submodule of the VISCY processor and was loaded by selecting it with the hierarchy browser of the application (see Chapter 7.6). This does not affect routing times, because the routing of a module is started when the module is selected. The indicators of the design are shown in Table 8.2. A screenshot can be seen in Figure A.2.

#Nodes	#Ports	#Standalone Ports	#Paths	Computed X/Y Constraint
62	315	85	135	616.11

Table 8.2: Design `VISCY-Controller` Indicators

8.1.1.3 Design: `wb_tx`

This design contains the transmitter of a Universal Asynchronous Receiver/Transmitter (UART) module created for the PicoNut processor. It is called `wb-tx` because the UART uses the Wishbone bus to connect the entire UART to the system [23]. The indicators of the design are shown in Table 8.3. A screenshot can be seen in Figure A.3.

#Nodes	#Ports	#Standalone Ports	#Paths	Computed X/Y Constraint
29	150	32	66	260.90

Table 8.3: Design `wb_tx` Indicators

8.1.1.4 Design: `PicoNut-Controller`

This design contains the controller of the PicoNut processor. It is a simple and modular 32-bit RISC-V processor developed by the EES group [23]. The indicators of the design are shown in Table 8.4. A screenshot can be seen in Figure A.4.

#Nodes	#Ports	#Standalone Ports	#Paths	Computed X/Y Constraint
103	511	130	222	979.30

Table 8.4: Design `PicoNut-Controller` Indicators

8.1.1.5 Design: `wb_rx`

This design contains the receiver of a UART module created for the PicoNut processor. It is called `wb-rx` because the UART uses the Wishbone bus to connect the entire UART to the system. It also has a submodule, which does not affect routing times because it is not routed until the submodule is selected. The indicators of the design are shown in Table 8.5. A screenshot can be seen in Figure A.5.

#Nodes	#Ports	#Standalone Ports	#Paths	Computed X/Y Constraint
64	284	47	110	462.60

Table 8.5: Design `wb_rx` Indicators

8.1.2 Results

The following sections contain the results of the three different cases that were described above. Each case contains times for all five designs. The measurements were repeated five times. Additionally, the average results are also provided in the tables.

8.1.2.1 Unmodified `libavoid`

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	0.21	0.22	0.21	0.21	0.21	0.21
VISCY-Controller	1.33	1.32	1.35	1.37	1.39	1.35
wb-tx	0.20	0.19	0.19	0.18	0.20	0.19
PicoNut-Controller	4.52	4.86	4.58	4.62	4.55	4.63
wb-rx	0.88	0.87	0.88	0.89	0.88	0.88

Table 8.6: Unmodifided `libavoid`: cola Runtime

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	1.08	1.07	1.07	1.06	1.08	1.07
VISCY-Controller	0.43	0.63	0.61	0.52	0.52	0.54
wb-tx	1.62	1.62	1.58	1.64	1.60	1.61
PicoNut-Controller	20.10	20.10	22.05	18.14	22.38	20.55
wb-rx	114.64	140.73	23.31	142.31	23.33	88.86

Table 8.7: Unmodifided `libavoid`: avoid Runtime

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	1.30	1.26	1.29	1.28	1.30	1.23
VISCY-Controller	1.77	1.96	1.97	1.89	1.91	1.90
wb-tx	1.82	1.82	1.78	1.82	1.81	1.81
PicoNut-Controller	24.64	24.96	22.64	22.78	26.96	24.40
wb-rx	115.53	141.61	24.20	143.20	24.22	89.75

Table 8.8: Unmodifided `libavoid`: complete Runtime

8.1.2.2 `libavoid` with modifications

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	0.21	0.21	0.20	0.22	0.21	0.21
VISCY-Controller	1.39	1.35	1.39	1.37	1.38	1.38
wb-tx	0.20	0.19	0.19	0.18	0.20	0.19
PicoNut-Controller	4.63	4.51	4.70	4.88	4.80	4.70
wb-rx	0.88	0.88	0.88	0.87	0.88	0.88

Table 8.9: Modified `libavoid`: cola Runtime

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	0.77	0.75	0.76	0.78	0.76	0.76
VISCY-Controller	0.40	0.63	0.61	0.52	0.52	0.54
wb-tx	1.14	1.14	1.11	1.15	1.14	1.14
PicoNut-Controller	16.53	16.59	16.61	2.55	15.32	13.52
wb-rx	88.05	87.40	106.96	108.50	85.39	94.66

Table 8.10: Modified `libavoid`: avoid Runtime

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	0.99	0.97	0.87	1.00	0.97	0.96
VISCY-Controller	1.80	2.01	1.80	1.63	1.65	1.78
wb-tx	1.34	1.34	1.30	1.33	1.35	1.33
PicoNut-Controller	21.19	21.12	21.33	7.44	20.13	18.24
wb-rx	89.05	88.30	107.76	109.38	86.28	96.15

Table 8.11: Modified `libavoid`: complete Runtime

8.1.2.3 `libavoid` in WebAssembly

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	0.34	0.48	0.59	0.49	0.48	0.48
VISCY-Controller	2.33	2.34	2.41	2.42	2.42	2.38
wb-tx	0.38	0.39	0.38	0.40	0.38	0.39
PicoNut-Controller	9.97	8.76	9.71	8.96	8.06	9.09
wb-rx	2.09	1.98	1.94	1.94	2.03	2.00

Table 8.12: Modified WebAssembly `libavoid`: cola Runtime

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	26.38	46.07	45.27	44.58	45.16	41.61
VISCY-Controller	1.20	1.21	1.22	1.23	1.22	1.21
wb-tx	10.22	14.01	10.22	7.17	14.07	11.14
PicoNut-Controller	35.75	37.54	18.78	57.30	37.37	37.75
wb-rx	0.31	0.33	0.32	0.42	0.33	0.34

Table 8.13: Modified WebAssembly `libavoid`: avoid Runtime

	Runtime in s					
Design	**1**	**2**	**3**	**4**	**5**	**Average**
regfile	26.74	46.61	45.78	45.08	45.66	41.97
VISCY-Controller	3.57	3.59	3.65	3.67	3.66	3.63
wb-tx	10.61	14.41	10.61	7.58	14.47	11.47
PicoNut-Controller	45.77	46.34	28.54	66.3	44.50	46.31
wb-rx	2.43	2.33	2.28	2.38	2.37	2.35

Table 8.14: Modified WebAssembly `libavoid`: complete Runtime

8.1.2.4 Graphs of the Messurements

The following graphs show the results from Chapters 8.1.2.1, 8.1.2.2, and 8.1.2.3. The runtimes used are the average values from the tables above. This was done with a Python script that was created with the help of ChatGPT. The axis used for the runtime is set to a logarithmic scale to improve the readability of the graphs. The data is grouped by each type of measurement, meaning that there is a graph for `libcola`, `libavoid`, and for the overall runtime.

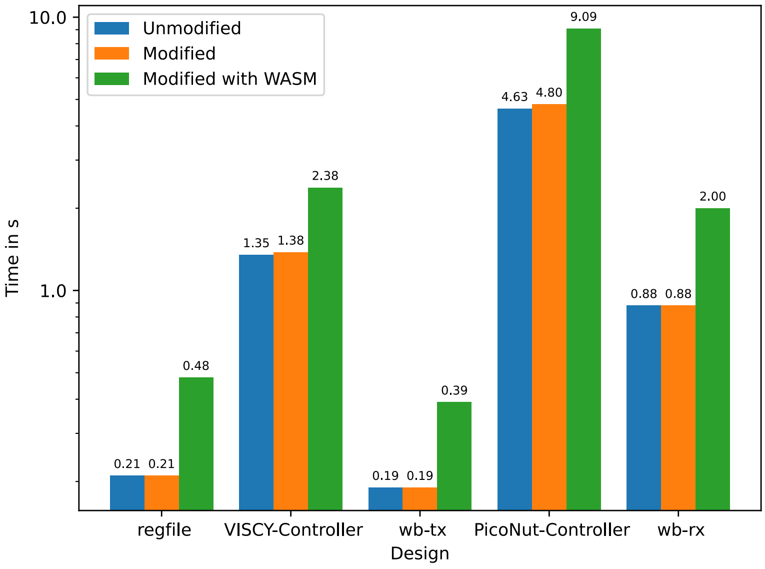


Figure 8.1: Graph of `libcola` Runtimes

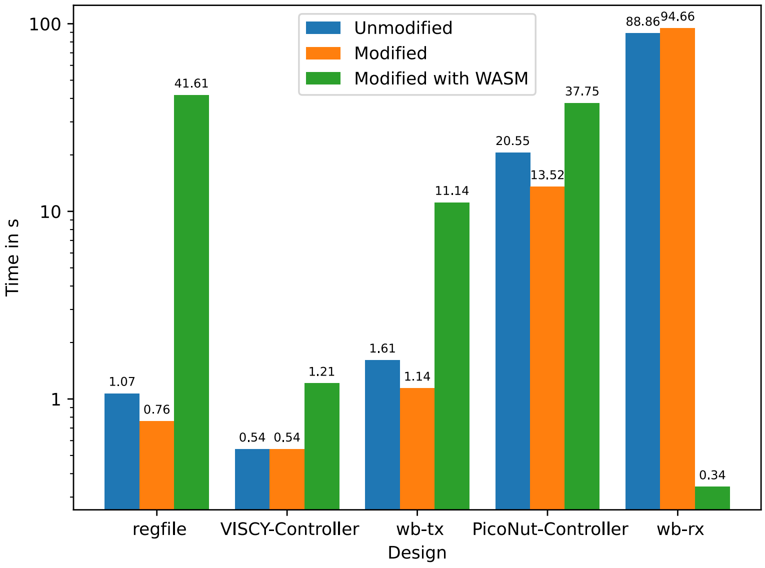


Figure 8.2: Graph of `libavoid` Runtimes

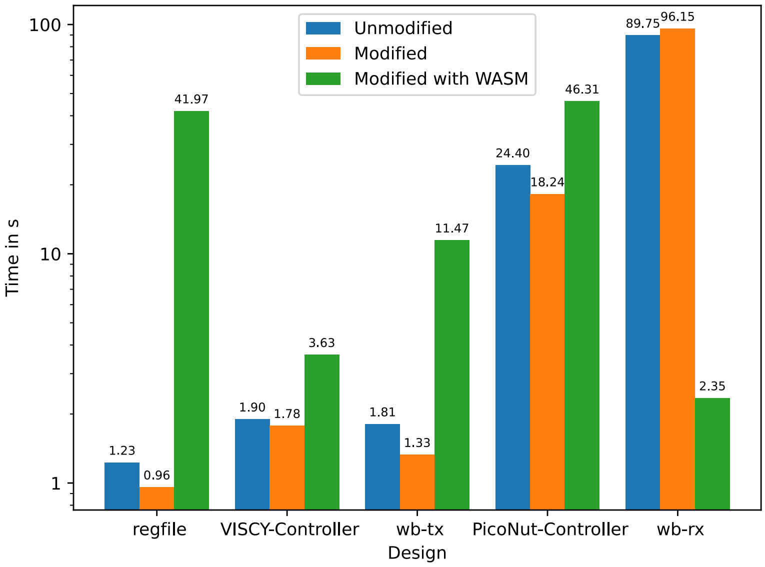


Figure 8.3: Graph of Overall Runtimes

8.1.3 Conclusion

When looking at the data in the tables in Chapter 8.1.2, the following conclusions can be drawn:

Firstly, the number of different objects in the designs has an impact on the runtime of the layouting process when the application is running natively on the system. This can be seen by comparing the tables with the design sizes in Chapter 8.1.1 and the runtimes in the tables of Chapters 8.1.2.1 and 8.1.2.2, as well as the figures in Chapter 8.1.2.4. This effect is not dependent on the modification of the `libavoid` library. However, there are also other factors that cannot be explained by the number of objects. This can be seen when comparing the object numbers of the ”PicoNut-Controller” and ”wb-rx”, seen in Chapters 8.1.1.4 and 8.1.1.5. Here, the ”PicoNut-Controller” has significantly more objects of all types, but the runtimes of the ”wb-rx” are between three and five times higher depending on the modification of `libavoid`. This might be due to the complexity of the connections within the design or the fact that many paths need to be routed similarly, causing the layouting of the paths to take considerably longer.

Furthermore, the runtime of the `libcola` library depends on the selected constraint values. As the X/Y constraints increase, the runtime of the layouting process also increases. This can be seen by comparing the constraint values in the tables of Chapter 8.1.1 and the runtimes in Figure 8.1.

When comparing the runtime increases of the WebAssembly build to both of the native builds, it can be seen that the WebAssembly build does not follow the same pattern as the native builds when it comes to the runtime of the routing with `libavoid`. These runtimes are sometimes higher and sometimes lower than the native values, as can be seen in Figure 8.1. This might be due to different behavior of the code when it is run in WASM. This difference does not apply to the times measured for `libcola` object routing. Comparing the numbers from the native runs (see Tables 8.6 and 8.9) with those of the WebAssembly build (see Table 8.12) shows that the values from the WASM build are approximately twice as high as the native ones. This can also be seen by looking at Figures 8.1 and 8.2.

For all of the test cases, the `libavoid` runtimes are significantly lower for some test runs than for others when testing a design for the same test case. Some examples are:

- `wb-rx` with unmodified `libavoid` in Table 8.7, where most of the runtimes are over 110s, but two are around 23s
- `PicoNut-Controller` with modified `libavoid` in Table 8.10, where the runtimes are mostly around 16s except for one which is 2.55s
- `PicoNut-Controller` with modified WebAssembly `libavoid` in Table 8.13, where the runtime varies between 18.78s and 57.30s

This shows that the routing process is not deterministic, which causes the runtimes to vary.

The runtimes of the `libcola` layouting process for all of the test cases are very stable. The runtimes for the native builds are very similar for all of the measured times. The runtimes of the WebAssembly build are also very stable across all the test runs but are higher, as mentioned before. This can be seen in Figure 8.1.

Comparing the unmodified and modified `libavoid` runtimes shows the following results, depicted in Table 8.15 and Figure 8.2. The results were calculated by taking the average of the runtimes and calculating the difference in percent between the two. Positive values in the table indicate a speedup of the modified version compared to the unmodified one, while negative values indicate a slowdown.

Design	unmodified in s	modified in s	speedup in %
regfile	1.07	0.76	29.0
VISCY-Controller	0.54	0.54	0.0
wb-tx	1.61	1.14	29.1
PicoNut-Controller	20.55	13.52	34.2
wb-rx	88.86	94.66	-7.7

Table 8.15: Speedup of modified `libavoid` Compared to Unmodified

Table 8.15 shows that the modification has a positive effect on the runtime of the `libavoid` library routing for some of the designs. The design `wb-rx` shows a slowdown in the average values, this is due to the two outliers in the runtimes of the unmodified version. When comparing the other runtimes of this design, the modified version is faster than the unmodified one, meaning that the modification has a positive effect on the runtime of the path routing process with `libavoid`.

When looking at the overall runtime, depicted in Figure 8.3, it can be seen that the times from the `libcola` routing (see Figure 8.1) and the `libavoid` routing (see Figure 8.2) roughly add up to the overall runtime. This means that, outside of the routing process, there is not much overhead in the application delaying the display of the selected design.

Looking at the combined runtimes in Figure 8.3 and the corresponding tables in Chapter 8.1.2 shows that the runtimes are too high for the size and complexity of the designs. In the worst case, the user in this test has to wait up to two minutes to see a schematic. This means that the application is not usable for larger and more complex designs. When examining the runtimes visualized in the tables and figures of Chapter 8.1.2, it becomes clear that the main factor contributing to the high runtimes is the path routing performed with `libavoid`. For the diagram with the highest runtime when running the application natively, ”wb-rx”, between 80 and 100 seconds on average are spent on path routing. This accounts for approximately 99% of the overall runtime. Therefore, improvements to the overall runtime can be achieved by optimizing the path routing process. One possible solution could be to replace the `libavoid` library’s algorithm with a more efficient one or to improve the existing implementation. More information on this can be found in Chapter 9.2.

Overall, it can be said that the runtimes of the `cola` library are more stable than those of the `libavoid` library. There, the runtimes are non-deterministic and can vary significantly between different runs. Additionally, in the WASM build of the application, the `avoid` path routing runtimes are not comparable with those of the native program.

8.2 WebAssembly Integration

8.2.1 Setup

This chapter presents the findings using the WebAssembly build of the application and the associated Qt library version. For this purpose, a system with the following specifications was used:

- Computer:
  - CPU: AMD Ryzen 7 5800X
  - RAM: 32 GB
  - GPU: NVIDIA GeForce RTX 3070
- Operating System: Debian 12.10 (April 2025)
- Browser (WASM): Google Chrome 134.0.6998.88
- Qt version: 6.8.1 and 6.7.3
- emscripten version: 3.1.56 (6.8.1) and 3.1.50 (6.7.3)
- Built in docker container
- Run on host (browser)

8.2.2 Results

8.2.2.1 Exceptions

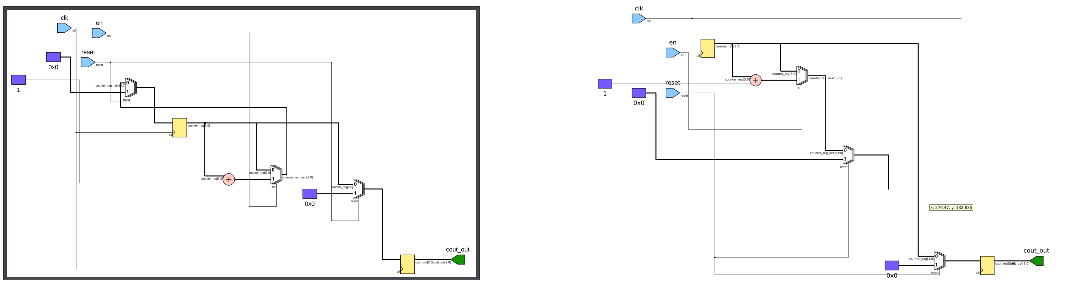
During the development of the application, a problem arose when trying to use exceptions in the WebAssembly build of the application. An error was thrown indicating that exceptions are not supported. The cause of this was that the prebuilt Qt library had been built without any support for exceptions. To fix this, the library was built from source using the instructions from the documentation [54]. This was integrated into the Docker container, more information can be found in Chapter 4.5.1.

8.2.2.2 Context Menus

At the beginning, the application used version 6.7.3 of the Qt library. This version of the library has a bug when using context or action menus. When clicking on an entry inside a submenu of a menu, the whole application crashes. This happens for all submenus, regardless of the way they were integrated into the GUI. This means it did not make a difference whether they were created with the help of Qt Designer or added in the source code of the application. This bug was fixed by upgrading the Qt library to version 6.8.1, indicating that it was a bug in the earlier version of the library. The problem was not present in the native version of the library.

8.2.2.3 Routing

During the creation of the routing time measurements for Chapter 8.1, a problem with the graphical output occurred. When the application was run in WebAssembly, the graphical output showed malformed lines when compared to the native version. Both versions of the program were generated from the same code base. This can be seen in Figures 8.4b and 8.4a. The output path of one of the paths does not connect to the input of the flip-flop.



(a) Native **(b)** WebAssembly

Figure 8.4: Comparison of routing WASM and native

This can be fixed by not running the `makeFeasible()` function of the `cola` router after the routing process. Although this prevents the malformed paths, it significantly increases the routing times of larger diagrams. The routing was tested with the designs from Chapter 8.1.1, but no times were recorded because it did not finish after multiple minutes.

Additionally, warnings thrown by the `avoid` routing algorithm are different in the WebAssembly build compared to the native version. For this, the `makeFeasible()` function is used, and the source code remains the same. The two are depicted in Listings 8.1 and 8.2.

Listing 8.1: Native `libavoid` Warning

```
xOffset value (30) in ShapeConnectionPin constructor greater than
    shape width (30).
```

Listing 8.2: WASM `libavoid` Warning

```
xOffset value (2384.84) in ShapeConnectionPin constructor greater than
     shape width (30).
```

The native warning is due to floating-point precision problems. The WebAssembly warning points to different behavior in the handling of floating point numbers. This is probably also the cause of the malformed lines.

8.2.3 Conclusion

The development of a WebAssembly version of an application that uses the Qt library is possible. Although the application works and is able to display schematics like the native version, it has some problems. The library itself can exhibit unexpected behavior and bugs that are not present in the native version. Additionally, differences between the toolchains can cause changes in functionality without the source code being changed. When a developer is working with WebAssembly and its version of the Qt library, they need to be aware that not all features may work as expected and that some workarounds might have to be implemented.

9 Conclusion

9.1 Summary

This thesis presents an approach to create an interactive schematic viewer that allows users to view digital circuits. The application is open-source and is able to read netlists in the Yosys-JSON format generated from various supported HDLs. By using the Qt framework in combination with WebAssembly technology, the application is able to run in a web browser, allowing users to test it without needing to install any software. Additionally, an AppImage version of the program is provided, enabling its use on a native Linux system without requiring any dependencies, while still benefiting from better performance compared to the WebAssembly version.

Furthermore, the application is able to calculate the positions of the components in the schematic using data from the parsed netlist files and symbols. It employs libraries from the Adaptagrams project, which use a constraint layout algorithm and avoidance routing to determine the final positions of the components and wires [3, 4]. The runtime of these libraries, especially `libavoid`, is too high for larger and more complex designs, even with improvements. This can lead to runtimes of several minutes, making the application unsuitable for such designs. Additionally, differences in the handling of floating-point numbers between the native and WebAssembly versions of the program can result in variations in both the runtime and the layout of the schematics.

The user interface allows the user to interact with the generated schematic to gain a better overview of the circuit. This includes the ability to zoom, pan, select, and highlight objects that are part of the schematic. It also allows displaying additional information about the selected objects. Additionally, the user can navigate through designs with multiple modules that are hierarchically arranged, either by clicking on them in the view, using a hierarchy tree, or by selecting the tabs of the different modules.

The program also provides options to configure the resulting schematic. For this, methods are provided to replace and customize the symbols used for the display. Additionally, the user can adjust the parameters used by the layout algorithm to improve the readability of the resulting schematics if the calculated values do not produce the desired result.

9.2 Outlook

In the current state of the application, the generation of schematics works well for small to medium-sized designs. However, the layout algorithm struggles with larger and more complex designs. Although some improvements have been made to increase the performance of the path layout algorithm by around 30%, it can still take several minutes to generate the schematic, meaning that the program is currently not usable in a productive setting. The user cannot estimate an exact runtime when opening a design. This could be improved by searching for further optimizations to the path routing algorithm. Another possibility would be to replace the currently used algorithm with a different one. For example, the algorithm behind the Eclipse Layout Kernel could be ported to C++ to allow its use in this project, or the Open Graph Drawing Framework could be extended to support the creation of objects with fixed ports.

Furthermore, the WebAssembly integration could be improved to prevent errors with the generation of schematics, such as the one shown in Chapter 8.2.2.3. This could potentially be achieved by using a more modern version of the Qt framework and, consequently, a newer version of the Emscripten compiler. Another possibility would be to adjust the code of the layout algorithms to work better within the WebAssembly environment.

Additionally, the application could be extended to support moving components within the graphical view. This would allow users to further adjust the output to their needs. It could also include a way to save the modified positions to a file so that users can reload their changes when starting a new session.

This could also be extended to allow the user to group signals together. It could be used, for example, to combine the signals of a bus (e.g., Wishbone) in order to reduce the number of paths in the schematic. As with the moving of components, these groups could be saved in a file to allow the user to reload them in a new session.

Another possible extension would be to add the list of nodes, paths, and standalone ports to the entry of each module in the hierarchy tree. This would allow the user to quickly see which objects the module contains. This also includes the ability to select the objects in the hierarchy tree and to select or highlight them in the schematic view, making it easier to navigate to a specific object in the graphical view.

Bibliography

[1] ADAPTAGRAMS: *Avoid Namespace Reference.* 2019. URL: https://www.adaptagrams.org/documentation/namespaceAvoid.html (visited on 2025-05-19) (cit. on p. 14).

[2] ADAPTAGRAMS: *cola Namespace Reference.* 2019. URL: https://www.adaptagrams.org/documentation/namespacecola.html (visited on 2025-04-18) (cit. on pp. 14, 47).

[3] ADAPTAGRAMS: *libavoid - Overview.* 2019. URL: https://www.adaptagrams.org/documentation/libavoid.html (visited on 2025-03-27) (cit. on pp. 13, 14, 77).

[4] ADAPTAGRAMS: *libcola - Overview.* 2019. URL: https://www.adaptagrams.org/documentation/libacola.html (visited on 2025-03-27) (cit. on pp. 13, 77).

[5] ADAPTAGRAMS: *libvpsc - Overview.* 2019. URL: https://www.adaptagrams.org/documentation/libvpsc.html (visited on 2025-03-27) (cit. on p. 13).

[6] ADAPTAGRAMS PROJECT: *adaptagrams.* URL: https://www.adaptagrams.org/ (visited on 2025-03-27) (cit. on pp. 12, 13, 47).

[7] ADVANCED MICRO DEVICES, INC: *Using the Schematic Window.* 2025. URL: https://docs.amd.com/r/en-US/ug906-vivado-design-analysis/Using-the-Schematic-Window (visited on 2025-03-31) (cit. on p. 1).

[8] BAUER, Lukas: *Yosysy HQ Conversation.* 2024-07 (cit. on p. 2).

[9] CHIMANI, Markus et al.: “The Open Graph Drawing Framework (OGDF).” In: *Handbook of graph drawing and visualization* 2011 (2013), pp. 543–569 (cit. on p. 12).

[10] DIGITALJS TEAM: *DigitalJs Online.* URL: https://digitaljs.tilk.eu/ (visited on 2025-03-28) (cit. on pp. 2, 20).

[11] DOCKER INC.: *What is Docker?* 2025. URL: https://docs.docker.com/get-started/docker-overview/ (visited on 2025-03-25) (cit. on p. 8).

[12] ECLIPSE FOUNDATION AISBL: *Eclipse Public License (EPL) Frequently Asked Questions.* URL: https://www.eclipse.org/legal/epl-2.0/faq/ (visited on 2025-03-27) (cit. on p. 11).

[13] ECLIPSE FOUNDATION INC.: *ELK Layered*. URL: `https://eclipse.dev/elk/reference/algorithms/org-eclipse-elk-layered.html` (visited on 2025-05-19) (cit. on p. 11).

[14] ECLIPSE FOUNDATION, INC: *Graph Data Structure*. URL: `https://eclipse.dev/elk/documentation/tooldevelopers/graphdatastructure.html` (visited on 2025-03-27) (cit. on p. 11).

[15] ECLIPSE FOUNDATION, INC.: *Documentation*. URL: `https://eclipse.dev/elk/documentation.html` (visited on 2025-03-27) (cit. on pp. 10, 11).

[16] ECLIPSE FOUNDATION, INC.: *Eclipse Layout Kernel*. URL: `https://eclipse.dev/elk/` (visited on 2025-03-27) (cit. on pp. 10–12).

[17] ELGRIS TECHNOLOGIES: *EDIF Overview*. URL: `https://www.elgris.com/content/edif_overview.html` (visited on 2025-03-20) (cit. on p. 4).

[18] EMSCRIPTEN CONTRIBUTORS: *About Emscripten*. 2015. URL: `https://emscripten.org/docs/introducing_emscripten/about_emscripten.html` (visited on 2025-03-21) (cit. on p. 6).

[19] EMSCRIPTEN CONTRIBUTORS: *Dynamic Linking*. 2015. URL: `https://emscripten.org/docs/compiling/Dynamic-Linking.html` (visited on 2025-04-01) (cit. on p. 23).

[20] EMSCRIPTEN CONTRIBUTORS: *emscripten*. 2015. URL: `https://emscripten.org/` (visited on 2025-03-21) (cit. on p. 6).

[21] FREE SOFTWARE FOUNDATION, INC: *Frequently Asked Questions about the GNU Licences*. 2024. URL: `https://www.gnu.org/licenses/gpl-faq.html` (visited on 2025-03-27) (cit. on p. 13).

[22] HOMEPAGE OF EES GROUP: *Efficient Embedded Systems*. 2025-03-05. URL: `https://ees.tha.de/` (visited on 2025-03-31) (cit. on p. 1).

[23] HOMEPAGE OF PICONUT PROJECT: *Welcome to the PicoNut Project*. 2025-03-05. URL: `https://ees.tha.de/piconut/index.html` (visited on 2025-03-31) (cit. on pp. 1, 65, 66).

[24] “IEEE Standard for VHDL Language Reference Manual”. In: *IEEE Std 1076-2019* (2019), pp. 1–673. DOI: `10.1109/IEEESTD.2019.8938196` (cit. on p. 4).

[25] INTEL CORPERATION: *Schematic View*. 2025. URL: `https://www.intel.com/content/www/us/en/docs/programmable/683641/21-3/schematic-view.html` (visited on 2025-03-31) (cit. on p. 1).

[26] JANSEN, Bernard J.: “The graphical user interface”. In: *SIGCHI Bull.* 30.2 (1998-04), pp. 22–26. ISSN: 0736-6906. DOI: `10.1145/279044.279051`. URL: `https://doi.org/10.1145/279044.279051` (cit. on p. 6).

[27] JEMALLOC: *jemalloc(3)*. Version 5.3.0-0-g54eaed1d8b56. 2023-06-01 (cit. on p. 57).

[28] JENKINS PROJECT: *Jenkins User Documentation*. URL: `https://www.jenkins.io/doc/` (visited on 2025-03-25) (cit. on p. 8).

[29] KIEFER, Gundolf: *Hardware-Systeme*. 2023-03 (cit. on pp. 5, 16, 65).

[30] KIELER: *elkjs*. URL: `https://github.com/kieler/elkjs` (visited on 2025-03-27) (cit. on p. 11).

[31] LATTICE SEMICONDUCTOR: *Diamond Overview*. 2025. URL: `https://www.latticesemi.com/en/Products/DesignSoftwareAndIP/FPGAandLDS/LatticeDiamond/DiamondOverview` (visited on 2025-03-31) (cit. on p. 1).

[32] LINUX MAN-PAGES: *brk(2)*. Version 6.03. 2022-12-04 (cit. on p. 57).

[33] LINUX MAN-PAGES: *malloc(3)*. Version 6.03. 2023-02-05 (cit. on p. 57).

[34] MDN WEB DOCS: *WebAssembly | MDN*. en-US. 2025-01. URL: `https://developer.mozilla.org/en-US/docs/WebAssembly` (visited on 2025-03-21) (cit. on p. 6).

[35] MJWYGROW: *adaptagrams*. URL: `https://github.com/mjwybrow/adaptagrams` (visited on 2025-03-27) (cit. on pp. 12, 14).

[36] NTURLEY: *netlistsvg*. URL: `https://github.com/nturley/netlistsvg` (visited on 2025-03-27) (cit. on pp. 1, 17, 19, 23).

[37] OPEN GRAPH DRAWING FRAMEWORK: *About*. URL: `https://ogdf.uos.de/about/` (visited on 2025-03-27) (cit. on p. 12).

[38] PATHANIA, Nikhil: *Learning continuous integration with Jenkins. A beginner's guide to implementing continuous integration and continuous delivery using Jenkins*. Community Experience Distilled. Includes bibliographical references and index. Packt Publishing, 2016. 542 pp. ISBN: 978-1-78528-483-0 (cit. on p. 8).

[39] QT WIKI: *About Qt*. 2022-07. URL: `https://wiki.qt.io/About_Qt#cite_ref-1` (visited on 2025-03-21) (cit. on pp. 6, 7).

[40] RUBIN, Steven M.: *Computer Aids for VLSI Design*. 1994. URL: `https://www.rulabinsky.com/cavd/text/chapd.html` (visited on 2025-03-20) (cit. on p. 4).

[41] SHERRICK, Susan: *An Introduction to Graphical User Interfaces and Their Use by CITIS*. en. 1992-1992-07-01. URL: `https://nvlpubs.nist.gov/nistpubs/Legacy/IR/nistir4876.pdf` (visited on 2025-03-21) (cit. on p. 6).

[42] TEROSHDL: *What is TerosHDL?* 2025. URL: `https://terostechnology.github.io/terosHDLdoc/docs/intro` (visited on 2025-03-28) (cit. on p. 19).

[43] TEROSTECHNOLOGY: *vscode-terosHDL*. URL: `https://github.com/TerosTechnology/vscode-terosHDL` (visited on 2025-03-28) (cit. on pp. 19, 40).

[44] THE APPIMAGE PROJECT: *AppImage. Introduction*. 2022. URL: `https://docs.appimage.org/index.html` (visited on 2025-03-25) (cit. on p. 9).

[45] THE APPIMAGE PROJECT: *AppImage. Motivation*. 2022. URL: `https://docs.appimage.org/introduction/motivation.html` (visited on 2025-03-25) (cit. on p. 9).

[46] THE APPIMAGE PROJECT: *AppImage. Concepts*. 2022. URL: `https://docs.appimage.org/introduction/concepts.html#` (visited on 2025-03-25) (cit. on p. 9).

[47] THE APPIMAGE PROJECT: *AppImage. linuxdeploy user guide*. 2022. URL: `https://docs.appimage.org/packaging-guide/from-source/linuxdeploy-user-guide.html` (visited on 2025-03-25) (cit. on p. 9).

[48] THE OGDF TEAM: *ogdf::OrthoLayout Class Reference*. 2024. URL: `https://ogdf.netlify.app/classogdf_1_1_ortho_layout.html` (visited on 2025-05-19) (cit. on p. 12).

[49] THE OGDF TEAM: *ogdf::PlanarizationLayout Class Reference*. 2024. URL: `https://ogdf.netlify.app/classogdf_1_1_planarization_layout.html` (visited on 2025-05-19) (cit. on p. 12).

[50] THE OGDF TEAM: *ogdf::SugiyamaLayout Class Reference*. 2024. URL: `https://ogdf.netlify.app/classogdf_1_1_sugiyama_layout.html` (visited on 2025-05-19) (cit. on p. 12).

[51] THE OGDF TEAM: *Open Graph Drawing Framework*. 2024. URL: `ogdf.netlify.app` (visited on 2025-03-27) (cit. on p. 12).

[52] THE QT COMPANY: *All Modules*. 2025. URL: `https://doc.qt.io/qt-6/qtmodules.html` (visited on 2025-04-01) (cit. on p. 23).

[53] THE QT COMPANY LTD.: *QFileDialog Class*. 2025. URL: `https://doc.qt.io/qt-6/qfiledialog.html#` (visited on 2025-04-10) (cit. on p. 32).

[54] THE QT COMPANY LTD.: *Qt for WebAssembly*. 2025. URL: `https://doc.qt.io/qt-6/wasm.html` (visited on 2025-03-21) (cit. on pp. 7, 30, 32, 74).

[55] THE QT COMPANY LTD.: *Qt Test Overview*. 2025. URL: `https://doc.qt.io/qt-6/qtest-overview.html` (visited on 2025-03-25) (cit. on pp. 9, 31).

[56] THE QT COMPANY LTD.: *Signals and Slots*. 2025. URL: `https://doc.qt.io/qt-6/signalsandslots.html` (visited on 2025-03-25) (cit. on p. 6).

[57] THE QT COMPANY LTD.: *The Qt Resource System*. 2025. URL: `https://doc.qt.io/qt-6/resources.html` (visited on 2025-03-25) (cit. on pp. 7, 40).

[58] THE QT COMPANY LTD.: *Using a Designer UI File in Your C++ Application*. 2025. URL: `https://doc.qt.io/qt-6/designer-using-a-ui-file.html` (visited on 2025-03-25) (cit. on p. 7).

[59] WEBASSEMBLY: *WebAssembly*. en. URL: `https://webassembly.org/` (visited on 2025-03-21) (cit. on p. 5).

[60] WEBASSEMBLY COMMUNITY GROUP: *Webassembly Specefication*. 2025-01. URL: `https://webassembly.github.io/spec/core/` (visited on 2025-03-21) (cit. on p. 5).

[61] YOSYSHQ: *oss-cad-suite-build*. URL: `https://github.com/YosysHQ/oss-cad-suite-build` (visited on 2025-03-21) (cit. on pp. 1, 5).

[62] YOSYSHQ GMBH: *prep - generic synthesis script*. 2025. URL: `https://yosyshq.readthedocs.io/projects/yosys/en/0.46/cmd/prep.html` (visited on 2025-03-28) (cit. on p. 16).

[63] YOSYSHQ GMBH: *show - generate schematics using graphviz*. 2025. URL: `https://yosyshq.readthedocs.io/projects/yosys/en/stable/cmd/show.html` (visited on 2025-03-28) (cit. on p. 1).

[64] YOSYSHQ GMBH: *What is Yosys*. 2025. URL: `https://yosyshq.readthedocs.io/projects/yosys/en/latest/introduction.html` (visited on 2025-03-20) (cit. on p. 5).

[65] YOSYSHQ GMBH: *write_json - write design to a JSON file*. 2025. URL: `https://yosyshq.readthedocs.io/projects/yosys/en/latest/cmd/write_json.html` (visited on 2025-03-20) (cit. on pp. 4, 5, 25).

I, Lukas Bauer, hereby declare that I have written the present thesis with the topic

> *Development of a Schematic Viewer for Digital Designs - as an Open-Source Project*

independently and have not used any sources or aids other than those indicated, and that I have marked all literal and analogous quotations as such. The thesis has not been submitted to any other examination authority and has not been published.

During the preparation of this thesis, the author used ChatGPT for spell and grammar checking. The author reviewed the changes proposed by the tool and edited the text as needed. He takes full responsibility for the content of this thesis.

Adelshofen, May 27, 2025

LUKAS BAUER

A Appendix

A.1 `libavoid` Patch

Listing A.1: Patch for `libavoid`

```
diff --git a/src/third_party/libavoid/vpsc.cpp b/src/third_party/libavoid/vpsc.cpp
index 530ce79..8c6dad9 100644
--- a/src/third_party/libavoid/vpsc.cpp
+++ b/src/third_party/libavoid/vpsc.cpp
@@ -1043,11 +1043,13 @@ void Block::deleteMinOutConstraint()
 }
 inline bool Block::canFollowLeft(Constraint const* c, Variable const* last) const
 {
-    return c->left->block == this && c->active && last != c->left;
+    const auto* left = c->left;
+    return left->block == this & c->active && last != left;
 }
 inline bool Block::canFollowRight(Constraint const* c, Variable const* last) const
 {
-    return c->right->block == this && c->active && last != c->right;
+    const auto* right = c->right;
+    return right->block == this && c->active && last != right;
 }

 // computes the derivative of v and the lagrange multipliers
@@ -1061,6 +1063,9 @@ double Block::compute_dfdv(Variable* const v, Variable* const u, Constraint*& mi
     for(Cit it = v->out.begin(); it != v->out.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowRight(c, u))
         {
             c->lm = compute_dfdv(c->right, v, min_lm);
@@ -1072,6 +1077,8 @@ double Block::compute_dfdv(Variable* const v, Variable* const u, Constraint*& mi
     for(Cit it = v->in.begin(); it != v->in.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
         if(canFollowLeft(c, u))
         {
             c->lm = -compute_dfdv(c->left, v, min_lm);
@@ -1088,6 +1095,9 @@ double Block::compute_dfdv(Variable* const v, Variable* const u)
     for(Cit it = v->out.begin(); it != v->out.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowRight(c, u))
         {
             c->lm = compute_dfdv(c->right, v);
@@ -1097,6 +1107,9 @@ double Block::compute_dfdv(Variable* const v, Variable* const u)
     for(Cit it = v->in.begin(); it != v->in.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
```

```
+
         if(canFollowLeft(c, u))
         {
             c->lm = -compute_dfdv(c->left, v);
@@ -1125,6 +1138,9 @@ bool Block::split_path(
     for(Cit it(v->in.begin()); it != v->in.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowLeft(c, u))
         {
 #ifdef LIBVPSC_LOGGING
@@ -1153,6 +1169,9 @@ bool Block::split_path(
     for(Cit it(v->out.begin()); it != v->out.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowRight(c, u))
         {
 #ifdef LIBVPSC_LOGGING
@@ -1232,6 +1251,9 @@ void Block::reset_active_lm(Variable* const v, Variable* const u)
     for(Cit it = v->out.begin(); it != v->out.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowRight(c, u))
         {
             c->lm = 0;
@@ -1241,6 +1263,9 @@ void Block::reset_active_lm(Variable* const v, Variable* const u)
     for(Cit it = v->in.begin(); it != v->in.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowLeft(c, u))
         {
             c->lm = 0;
@@ -1253,6 +1278,9 @@ void Block::list_active(Variable* const v, Variable* const u)
     for(Cit it = v->out.begin(); it != v->out.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowRight(c, u))
         {
 #ifdef LIBVPSC_LOGGING
@@ -1265,6 +1293,9 @@ void Block::list_active(Variable* const v, Variable* const u)
     for(Cit it = v->in.begin(); it != v->in.end(); ++it)
     {
         Constraint* c = *it;
+        if(!c->active)
+            continue;
+
         if(canFollowLeft(c, u))
         {
 #ifdef LIBVPSC_LOGGING
@@ -1321,11 +1352,17 @@ void Block::populateSplitBlock(Block* b, Variable* v, Variable const* u)
     b->addVariable(v);
     for(Cit c = v->in.begin(); c != v->in.end(); ++c)
     {
+        if(!((*c)->active))
+            continue;
+
         if(canFollowLeft(*c, u))
             populateSplitBlock(b, (*c)->left, v);
     }
     for(Cit c = v->out.begin(); c != v->out.end(); ++c)
     {
```

```
+        if(!((*c)->active))
+            continue;
+
         if(canFollowRight(*c, u))
             populateSplitBlock(b, (*c)->right, v);
     }
@@ -1339,6 +1376,9 @@ bool Block::getActivePathBetween(Constraints& path, Variable const* u, Variable
         return true;
     for(Cit_const c = u->in.begin(); c != u->in.end(); ++c)
     {
+        if(!((*c)->active))
+            continue;
+
         if(canFollowLeft(*c, w))
         {
             if(getActivePathBetween(path, (*c)->left, v, u))
@@ -1350,6 +1390,9 @@ bool Block::getActivePathBetween(Constraints& path, Variable const* u, Variable
     }
     for(Cit_const c = u->out.begin(); c != u->out.end(); ++c)
     {
+        if(!((*c)->active))
+            continue;
+
         if(canFollowRight(*c, w))
         {
             if(getActivePathBetween(path, (*c)->right, v, u))
@@ -1370,6 +1413,9 @@ bool Block::isActiveDirectedPathBetween(Variable const* u, Variable const* v) co
         return true;
     for(Cit_const c = u->out.begin(); c != u->out.end(); ++c)
     {
+        if(!((*c)->active))
+            continue;
+
         if(canFollowRight(*c, nullptr))
         {
             if(isActiveDirectedPathBetween((*c)->right, v))
@@ -1387,6 +1433,9 @@ bool Block::getActiveDirectedPathBetween(
         return true;
     for(Cit_const c = u->out.begin(); c != u->out.end(); ++c)
     {
+        if(!((*c)->active))
+            continue;
+
         if(canFollowRight(*c, nullptr))
         {
             if(getActiveDirectedPathBetween(path, (*c)->right, v))
```

A.2 Screenshots of Used Diagrams

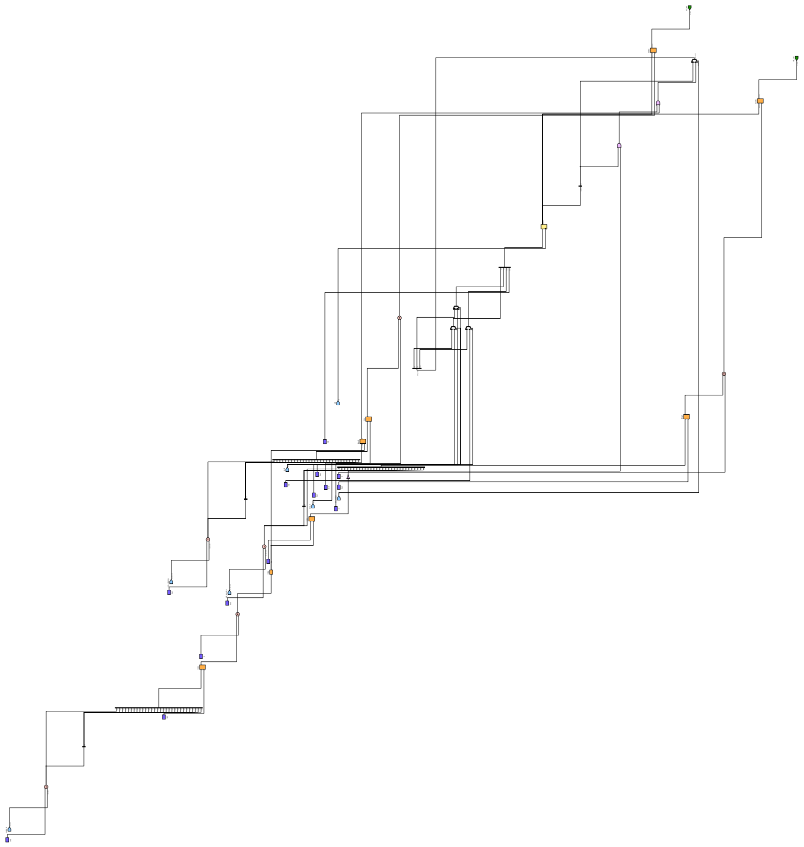


Figure A.1: Regfile Screenshot

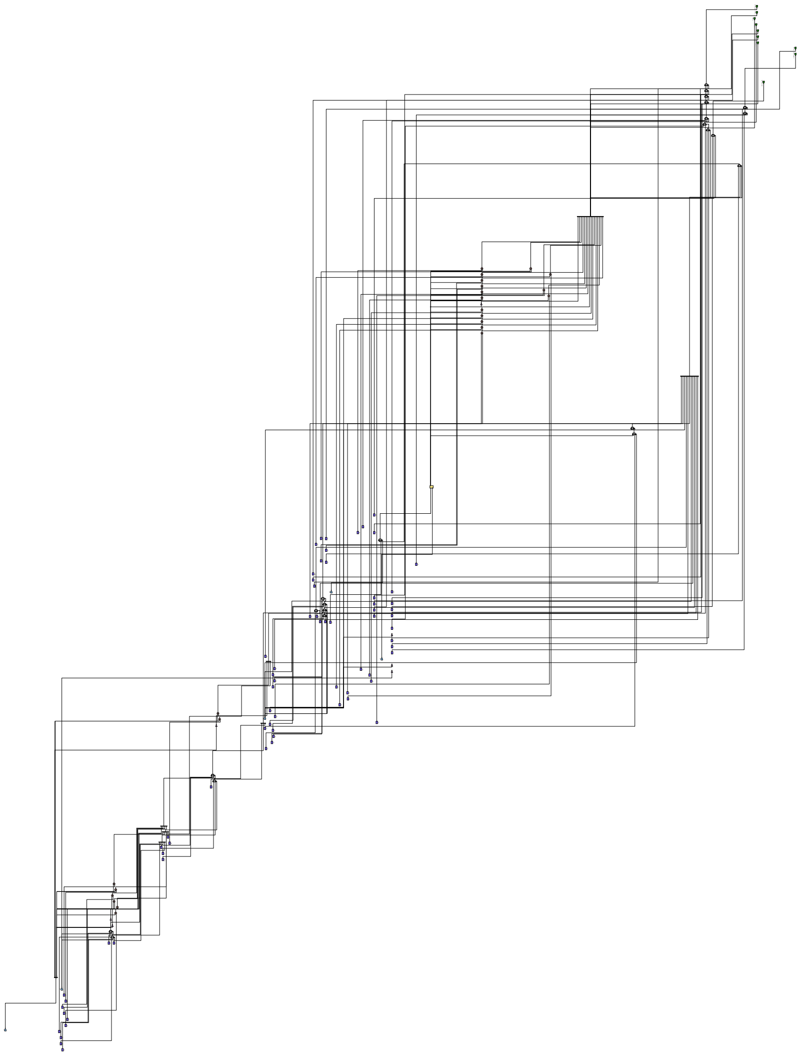


Figure A.2: VISCY-Controller Screenshot

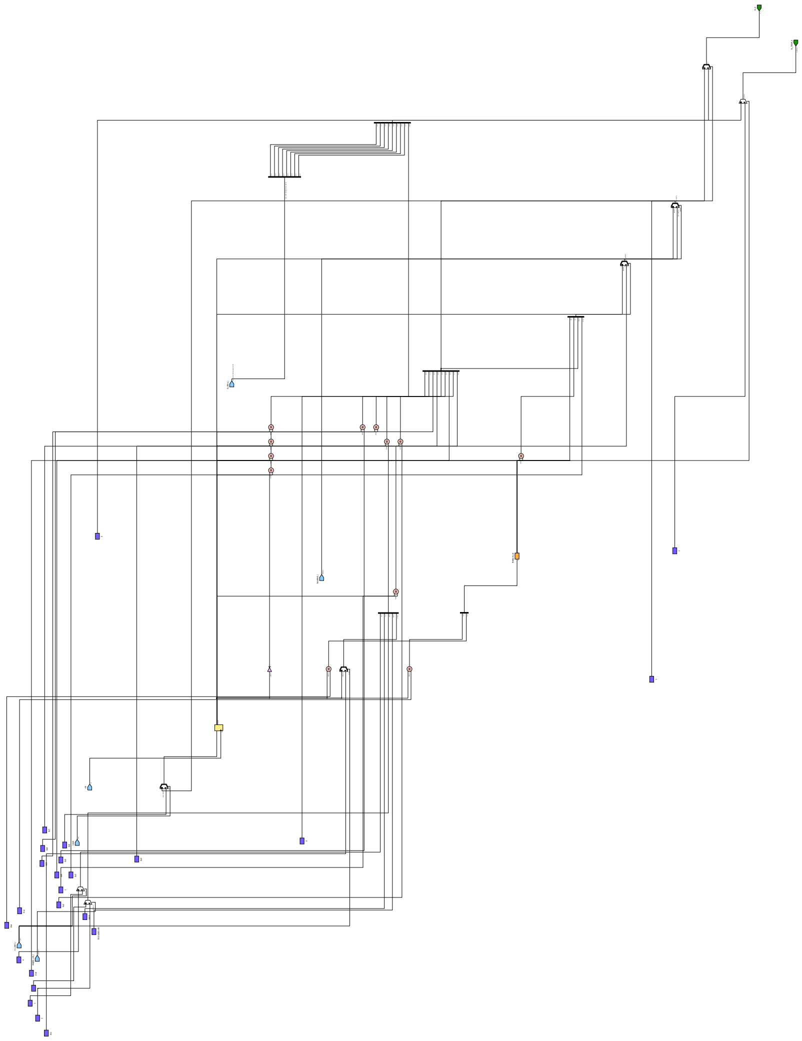


Figure A.3: WB-TX Screenshot

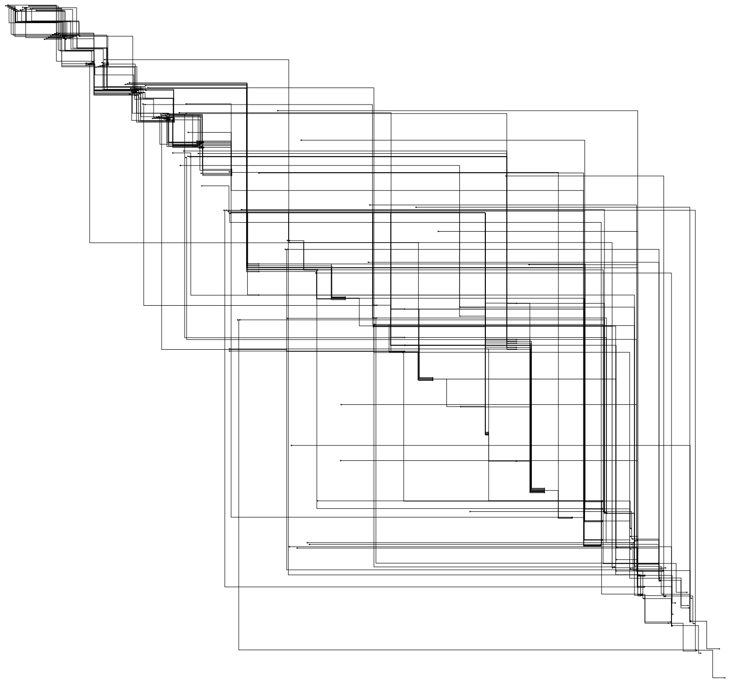


Figure A.4: Pico-Controller Screenshot

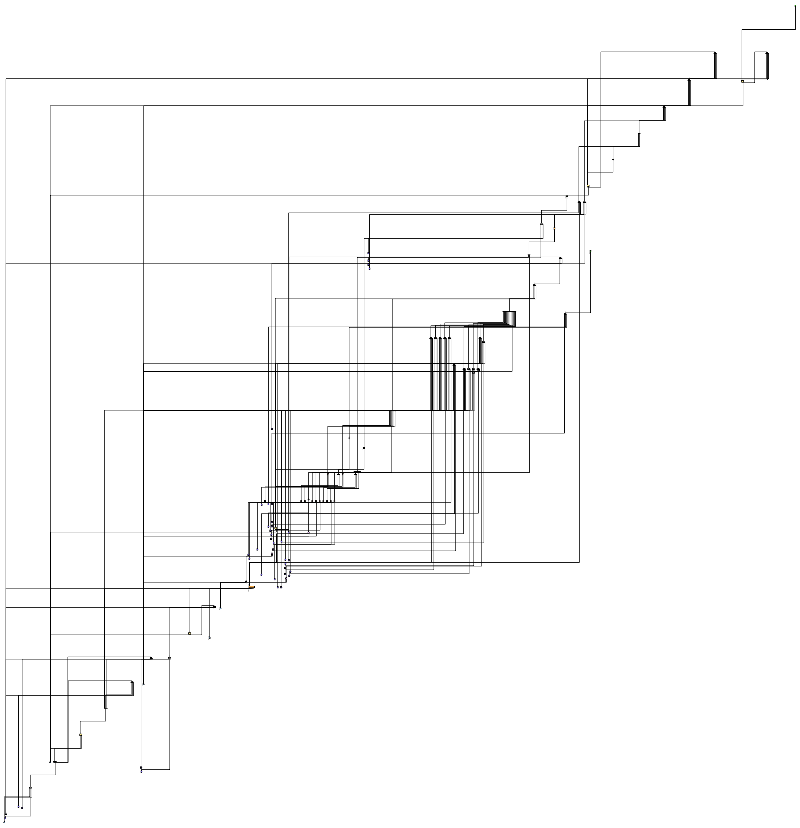


Figure A.5: WB-RX Screenshot