

NandSim

Digital logic circuit simulator

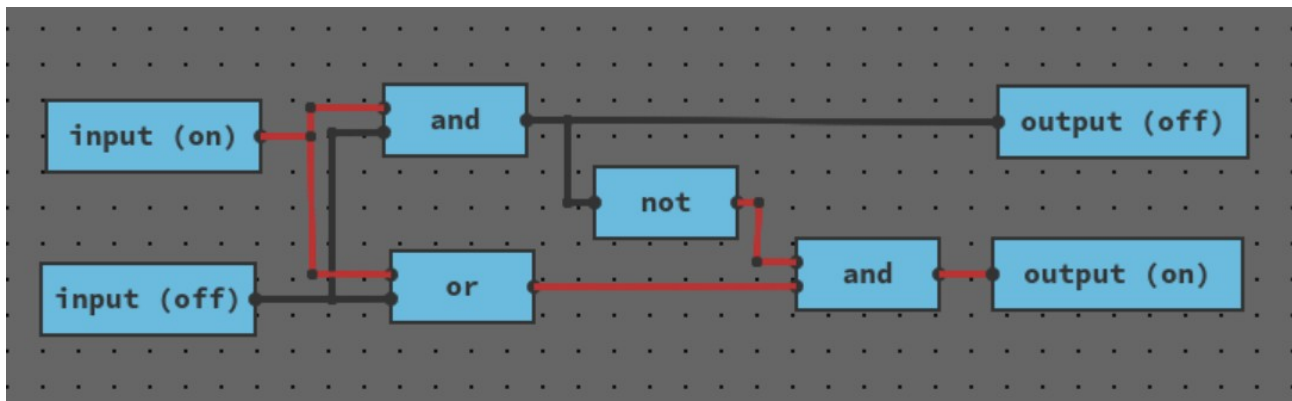


Table of Contents

Terminology.....	5
Introduction.....	6
Case description (case-beskrivelse).....	9
Case 1: Demonstrate the behavior of a 4-bit full adder.....	10
Case 2: Demonstrate the correctness of a piece of microcode for a 7-segment display encoder. .	10
Requirement specification (kravspecifikation).....	12
Requirements.....	12
Non-functional requirements.....	13
Delimitation.....	14
User manual.....	15
Theory.....	15
Tour.....	17
Circuit editor.....	18
Controlling the camera.....	18
Selecting components.....	18
Moving items.....	19
Deleting items.....	20
Activating inputs.....	20
Wiring.....	20
Toolbar.....	21
Place components.....	22
Open component editor.....	23
Tab-bar.....	23
Create components.....	23
Component control buttons.....	24
Rename component.....	24
Save component.....	24
Close editor tab.....	24
Example – An XOR component.....	24
Product documentation.....	30
The concept.....	30
Digital twin.....	30
Visual and interactive.....	31
Satisfying requirements.....	31
Building circuits.....	32
Create components.....	33
Simulate circuit.....	34
Upload program and data files.....	34
Solution (Problembeskrivelser).....	35
Graphical desktop application.....	35
Events.....	38
Editor.....	42
Project.....	44
Board.....	46

Simulation.....	47
IR.....	48
Circuit lowering.....	53
Optimizing IR.....	54
Simulation.....	57
UI, IO, and rendering.....	58
Sequence diagrams.....	60
Class diagram.....	62
Literature.....	63

Table of Figures

Figure 1: Part of the process of introducing an IC on the market, involving the steps 'Design hardware', 'Produce hardware' and then 'Implement firmware'.....	8
Figure 2: The same process, but with the steps 'Produce hardware' and 'Implement firmware' being in parallel.....	9
Figure 3: A screenshot of NandSim.....	16
Figure 4: Screenshot of NandSim with arrows showing each element.....	19
Figure 5: One component selected.....	20
Figure 6: Multiple components and joints added to selection.....	20
Figure 7: Selecting multiple items with box selection.....	21
Figure 8: Circuit with inputs activated.....	22
Figure 9: Hovering the mouse over a pin.....	22
Figure 10: Two components being connected.....	23
Figure 11: Joints in wiring mode.....	23
Figure 12: Toolbar initially.....	24
Figure 13: Toolbar with more components.....	24
Figure 14: Toolbar with selected component.....	24
Figure 15: Outline shown in editor of component selected in toolbar.....	25
Figure 16: Tab-bar with multiple tabs.....	25
Figure 17: XOR gate truth table.....	26
Figure 18: Empty tab.....	27
Figure 19: XOR internal components placed.....	28
Figure 20: XOR components wired together.....	29
Figure 21: XOR all input combinations tested with corresponding output.....	30
Figure 22: Save and Rename buttons highlighted.....	30
Figure 23: Circuit using XOR components.....	31
Figure 24: Sketch of toolbar and editor-area.....	34
Figure 25: ROM editor pop-up window.....	37
Figure 26: High-level class diagram showing the relation between Editor, EventBus and ViewPos	44
Figure 27: High-level class diagram showing the State pattern.....	46
Figure 28: Class diagram of Board.....	49
Figure 29: In-editor representation of trivial circuit.....	51
Figure 30: IR representation of trivial circuit.....	51
Figure 31: Example of circuit of XOR component.....	52
Figure 32: Class diagram of IR.....	53
Figure 33: Simple input-output circuit with 3 joints.....	56
Figure 34: Input-output circuit IR before and after optimizing.....	57
Figure 35: Difference between optimizeMain and optimizeComponent.....	58
Figure 36: Sequence diagram when the application initializes.....	62
Figure 37: Sequence diagram of a mouse click that places a component and subsequently triggers a re-render.....	63
Figure 38: Class diagram.....	64

Terminology

Throughout the report, including the case description and requirement specification, I use the terms firmware, embedded software, and microcode large interchangeably. In all cases I mean software developed by the IC manufacturer.

Introduction

The process of designing and manufacturing integrated circuit chips (ICs) involves significant commitments of time and resources. The process of introducing a new chip on the market is long and complicated. It involves many steps and sub-processes. The majority of these steps and sub-processes are contingent on each other. This includes temporal coupling of different stages of the development process. In layman's terms this means that one department is hard at work, while another department is without work, waiting for the first department to reach some milestone, where the second department's ability comes into relevancy. Loosening the coupling of departments like these will decrease the overall development time of a chips, improve time to market, and reduce the needed commitment of time and resources. This report details a solutions seeking to do just that, by employing a software application.

Part of the process of introducing a new chip on the market is developing firmware. Customers of ICs need to be able to integrate the chips with their systems in ways that fit their use-cases. Developing software for an IC can be a complicated endeavor requiring deep knowledge of how the hardware functions. Part of the package when buying an IC is a layer of software that exists between the user's software and the hardware. This layer, called firmware, is most advantageously developed by the IC manufacturer, as they have the deepest knowledge of the hardware. Therefore, developing this firmware is a part of the overall development process.

There are three specific parts of the aforementioned process that are relevant to this report. It pertains the designing of the hardware, producing the hardware, and implementing firmware for the hardware. Leaving out non-technical work, the first subprocess of introducing a new chip on the market is designing the hardware of the chip itself. At some milestone late in the process of designing the hardware, the schematics and specifications are sent off to a manufacturing plant where the newly designed chips get produced. At some later point, hardware samples will be shipped back the company.

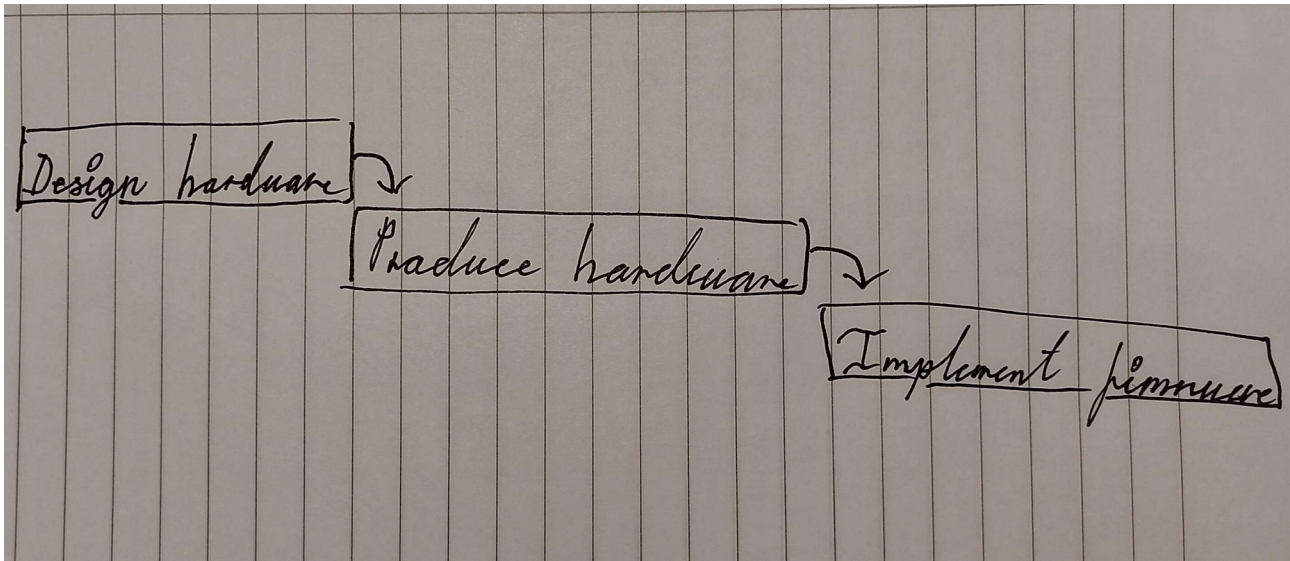


Figure 1: Part of the process of introducing an IC on the market, involving the steps 'Design hardware', 'Produce hardware' and then 'Implement firmware.'

Conventionally, it is first at this later point when hardware has been shipped back to the company, that the firmware development department can begin developing the firmware. The reason for this, is that developing the firmware requires testing on the actual hardware. It is for this reason that firmware development is tightly coupled sequentially to hardware production.

The goal of this project is to make that decoupling of hardware production and firmware development. By decoupling these two, the two subprocesses can be initiated in parallel. As the work of these two subprocesses are done by different personnel, the overall project time can be reduced in this manner.

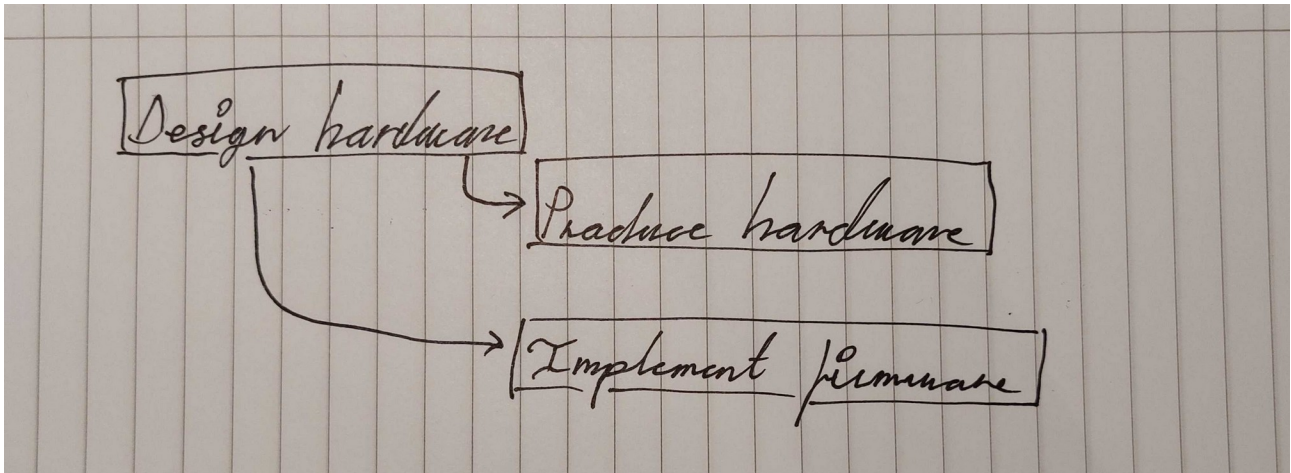


Figure 2: The same process, but with the steps 'Produce hardware' and 'Implement firmware' being in parallel.

By utilizing a software tool to simulate a *digital twin* of the actual designed hardware, that is able to emulate the hardware to a sufficient degree, firmware development can be initiated already when this digital twin is realized. A digital twin in a simulation environment requires no production of physical hardware and there's thus not the same degree of dependence on external organizations. Developing a digital twin can be done in a more time efficient manner than obtaining physical hardware, primarily for the reason that both producing the hardware takes time, but also the shipping of hardware between the location of the development departments and the manufacturing plant.

This report will detail exactly this piece of software, a digital logic circuit simulator.

Case description (case-beskrivelse)

The following case description has been developed in collaboration with the customer prior to initiating the project:

Logic simulator for integrated circuits

We are Kim's Chips (Intel, AMD, Atmel, ST Microcontrollers, etc.). As an integrated circuit manufacturer we develop and produce integrated circuit solutions. These solutions consist both of physical hardware in the form of silicon chips and embedded software.

Integrated circuits are electrical circuits built into silicon chips. Silicon chips use semiconductor technology and are manufactured using lithography technology. In addition to the hardware, integrated circuits are also sold with accompanying embedded software. The embedded software provides support, so that the customers can write their specific applications on top.

The development process of integrated circuits consist there both of designing the actual hardware, but also development of the embedded software. An inefficiency arises in the development process. The issue is that development of the embedded software has to take place after both design and production of the physical hardware. This is because the software can't be run or tested without the hardware. This makes the software development process dependent on the hardware production process. Because of this, the software development process cannot be done in parallel with the hardware development or production, it has to be done afterwards. It would be beneficial, if this process could be done in parallel.

Logic circuits are circuits of logic gates. Logic circuits can be used as an abstraction of electrical circuits for integrated circuits. For complex chip designs, logic circuits are used as the primary medium for designing the chip. In addition to being used for designing the integrated circuit, a logic circuit can also be used for simulations.

Because logic circuits can be used for simulations, they can also enable testing of embedded software. If you therefore can build a chip as a logic circuit, you can simulate the chip, and run and test software on the simulated chip. Using this method, the embedded software development process can be decoupled as to be independent from the hardware development and production process.

For this purpose, we would like a software solution that enables us as chip manufacturers to design and specify logic circuits as to simulate hardware chips to run embedded software. The circuits should be specified by assembling logic gates. We would like a visual tool for assembling and organizing logic gates. The tool should be able to simulate a specified circuit. We want the simulation to be a live simulation, where we can interactively test software and interact with the circuit.

Case 1: Demonstrate the behavior of a 4-bit full adder

Chip development involved designing the schematic of various complex logic components. One such component could be a 4-bit full adder. The simulator needs to be able to demonstrate that the behavior of a specific schematic for complex component, in this case a 4-bit full adder, is correct and will work in practice.

Given a schematic of a 4-bit full adder, the system has to provide functionality to wire together components according to the schematic. When the circuit wiring is done, the system has to simulate the behavior of the component correctly and interactively. The simulator should demonstrate that the 4-bit full adder can add numbers correctly, for example that $3 + 5 = 8$.

Case 2: Demonstrate the correctness of a piece of microcode for a 7-segment display encoder

Often, a chip needs certain programs of microcode. Microcode are pieces of code or data that are used internally in chips. A schematic of a 7-segment display encoder can utilize a ROM component for the encoding itself. ROM (read-only memory) components, and memory components generally, function by associating address bits

with data bits. A 7-segment display (in this instance) takes as input the state of each of the 7 segments.

Using 2 4-bit ROM components (4-bit address and value) an encoder can be made that translates the binary bit values of the integers 0 to 15 to their corresponding 7-segment pin combinations. The system has to provide functionality for such a component to be designed so that microcode can be tested on it.

Requirement specification (kravspecifikation)

NandSim – A tool for specifying and simulating integrated circuits with logic gates.

The customer is an integrated circuit chip (IC) manufacturer. The tool is to be designed used for software developers and as an interaction point between hardware developers and software developers.

The solution is a visual tool, that both software and hardware developers can use to build and simulate logic circuit models. The models consist of assembled logic gates. To organize complicated logic, the gates of the circuits can be organized into reusable component. The tool can simulate the modeled circuits in an interactive live simulation. Software developers should be able to develop software targetting the modeled circuits, and further be able to execute and test code in the simulation.

The goal is a tool that enable IC manufacturers, specifically software and hardware developers to design and develop circuit models and embedded software, as to reduce the development time for an IC solution, by decoupling the software and hardware development processes.

Requirements

As an IC developer using this solution, I should be able to:

- **★ assemble logic circuits visually by placing and wiring together primitive logic gates.** This includes a visual editor area, where logic gates and components can be inserted and dragged around, and where each gate and component can be wired together.
- **★ create circuit components built with logic gates that can be used like primitive logic gates.** To help organize and make complex circuits manageable, sub-sections of a circuit can be encapsulated in a component.
- **★ simulate the logic of a circuit.** A simulation consists of initializing a state according to the circuit and the inputs given. The simulation should be continuous, and support simulating state dependent on a previous state.
- **★ interact visually with a simulation of a circuit.** The simulation should support updating the state according to any changes made to the circuit.

- **Upload files with data input for complex circuits, including program and data files in suitable file formats such as binary or hexadecimal.** The purpose is to enable flashing of a simulated circuit with data and programs to be used in the simulation.
- **Monitor complex circuit output in suitable formats such as base 10 displays, hexadecimal or ASCII text.** To be able to monitor the produced outputs of a simulated circuits, the tools should have ways to display outputs appropriately.
- **Create and export component libraries.** Components created in one project should also be usable in other projects.
- **Save projects in the cloud with a user account.** Projects can be stored in the cloud using user accounts. With the same user account, projects can be accessed, edited and saved on multiple computers.
- **Collaborate with others using cloud projects and multiple user accounts.** Multiple people can collaborate on the same project, each using their own user account. This includes a strategy for work synchronization and work conflict resolution.

★ marked requirements demarcate the minimum viable product.

Non-functional requirements

The system should conform to the non-functional requirements of:

- **Simulating interactively without unreasonable delay.** The system should be able to simulate any circuit with less than 100 components in less than 500 milliseconds on modern hardware.
- **Preventing loss of work.** The system should employ a strategy to save the user's progress frequently enough, so that work is never lost. Immediately following any significant change, the system should be able to restart without loss of work.

Delimitation

For simplicity, both of use and of implementation, the solution will be a digital logic simulator. This is an abstraction removed from simulating electrical current in circuits with resistors, capacitors, etc. This limits the tool in terms of it's applicability while still being suited for the case description.

The simulator will not handle unstable circuits. If a circuit contains a cycling connection in such a way that the circuit vacillates indefinitely, the circuit is said to be unstable. A general solution to detect unstable circuits ahead of time is mathematically impossible due to the halting problem, and heuristics-based detection methods come with their own complexity. It's decided that the simulator simply doesn't support unstable circuits, and it's thus the user's responsibility ensure that circuits are stable.

User manual

The application, NandSim, is a digital logic simulator. The purpose of the application is to allow the user to construct and simulate digital logic circuits using built-in and custom defined components.

The application is a web application accessible publicly on the domain <https://nandsim.sfja.dk/>.

When opening the website in a browser, the user is presented with the following:

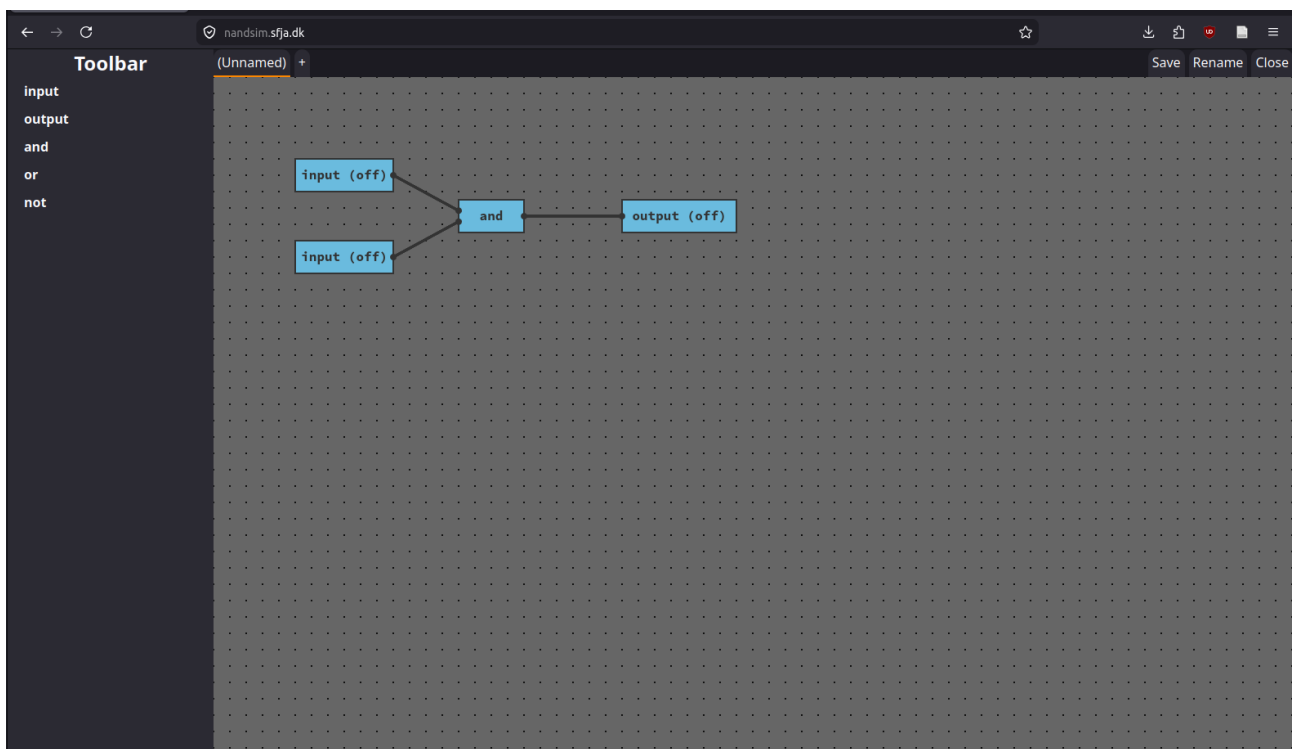


Figure 3: A screenshot of NandSim

The screenshot in Figure 3 shows the application when initially opened. As can be seen the application starts with an initial circuit in the circuit editor in a tab called '(Unnamed)'. There are 5 components predefined in the toolbar: 'input', 'output', 'and', 'or', and 'not'.

Double clicking either of the 'input' components switches their state from 'off' to 'on'. Doing so will activate the wires connected to the components. Activating both 'input' components in the initial example circuit, will activate the 'output' component.

Theory

The application is a digital logic circuit simulator. *Digital logic*, or boolean algebra, is a branch of mathematics that deals with truth values (*true* and *false*). The algebra uses logical operators to build

up expressions based on these truth values to express new values. Fundamentally, the logical operators are *conjunction* (AND), *disjunction* (OR), and *negation* (NOT). In mathematics, the symbols ' \wedge ', ' \vee ' and ' \neg ', and in programming symbols similar to '&', '|' and '!', are used to represent these logical operators. Expressions can be built using these primitives, and expressed such as $true \wedge \neg(false \vee true)$.

Digital logic uses these principles in a more practical sense, where *true* and *false* may represent the values 0 and 1, and where boolean expressions can be used to represent higher mathematics constructs.

Electrical circuits are circuits in-which electrical current passes through electrical components connected with wires. Using specific combinations of electrical components, namely transistors or components with similar switching functionality, electrical circuits can be used to simulate digital logic. This is fundamentally how digital computer chip function. By representing *true* or 1 as positive and *false* or 0 as connection to ground or negative, electrical circuits can simulate digital logic.

Digital logic circuits are an abstraction of electrical circuits. Electrical circuits, given that they're using electricity, has to deal with the laws and intricacies of electricity. Electricity functions with voltage and current, and electrical circuits have to manage the flow of current correctly. Digital logic circuits abstract away these laws and intricacies that are irrelevant to the digital logic itself. In digital logic, there are two states, *true/high/activated/on* and *false/low/deactivated/off*. Digital logic circuitry therefore only deals with these two states.

Digital logic circuits consists of components with inputs and outputs, and wires connecting these components. Wires can transmit either an *on*-signal or an *off*-signal. Complex digital logic can thus be implemented by connecting components simulating logic operators.

The circuits in NandSim consists of three fundamental elements: *components*, *wires* and *joints*. *Components* are the logical components discussed above. Components consists of a *label* or name, *input pins* and *output pins*. *Wires* connect each component's input and output pins to other component's input and output pins. *Joints* connect more wires together. Joints transfers the signals from each wire to all other connected to the joint. Components and joints can be placed and moved in positions relative to the circuit.

Tour

Here the various elements that makes up the application can be seen:

1. **Circuit board editor.** This is the area where components are placed and wired together. Circuits are simulated interactively. When any change occurs to the circuit, the simulation is updated.
2. **Toolbar.** The toolbar shows a list of every available component. To place components on the editor area, you first select the tool the toolbar. When new components are created and saved, they appear in the toolbar as usable components.
3. **Tab-bar.** Separate components are created in separate tabs. Each open component editor will appear in the tab-bar. The '+' button can be used to open new tabs, to create new components.
4. **Component control buttons.** The component control buttons are used to manage, the components and tabs of your project.

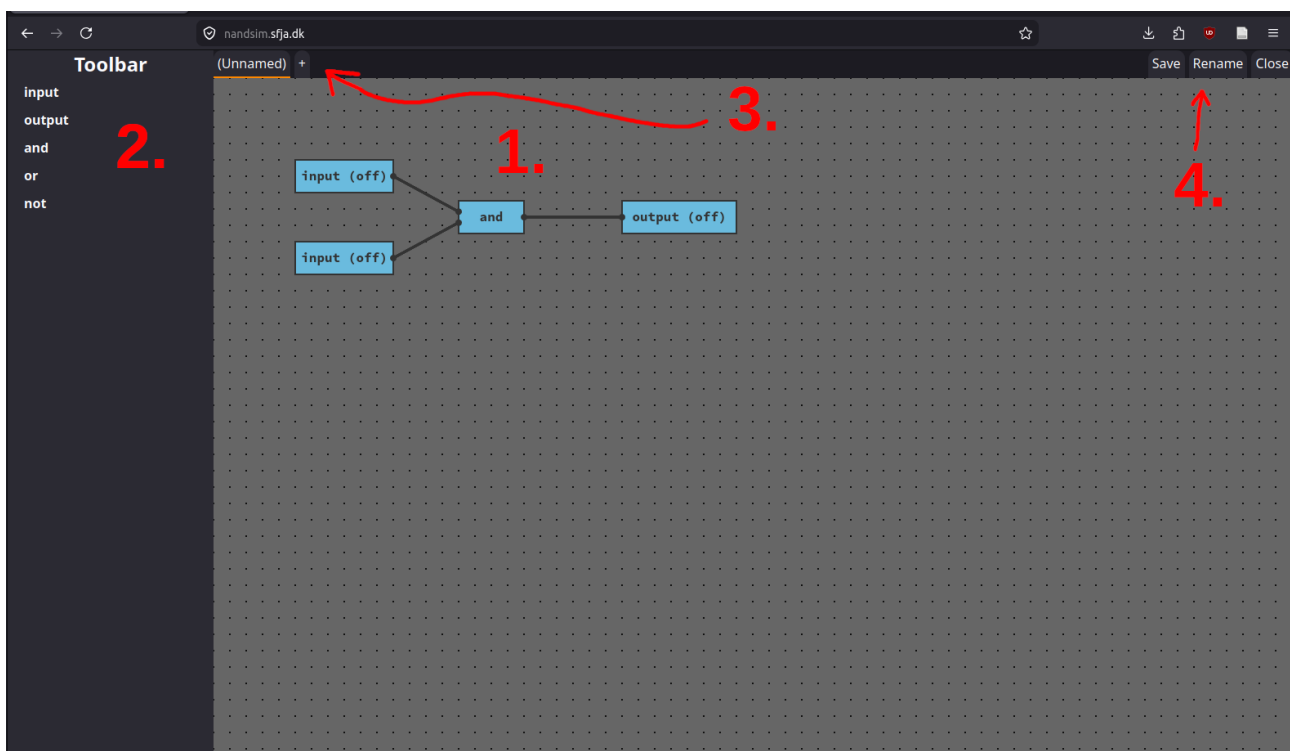


Figure 4: Screenshot of NandSim with arrows showing each element.

Circuit editor

The circuit editor is main element of the program. It has many features that allow the user to create and edit circuits, and interact with the circuit simulation interactively.

Controlling the camera

The editor area is an infinite grid where components can be placed. The grid is seen top-down with a given initial position. The *camera* point down on the grid can be moved laterally. By pressing the 'Shift' key while clicking and dragging the grid, the entire grid will move with the mouse. The camera position will reset when switching tabs or refreshing the page.

Selecting components

By clicking a component, the component will become selected. This is indicated by an orange outline.

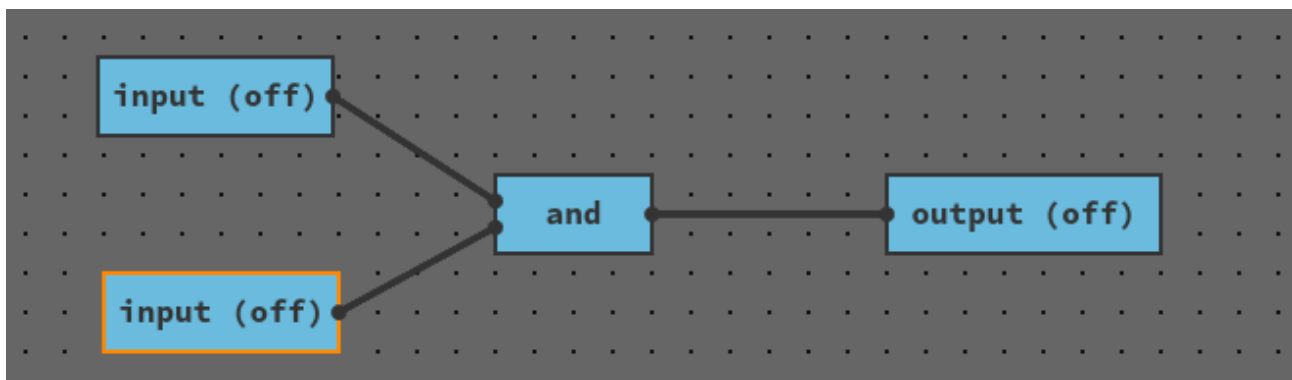


Figure 5: One component selected

Remove the selection by clicking empty grid area. The orange outline will disappear.

After selecting component, additional components and wire-joints can be toggled in and out of the selection by holding the 'Control' key and clicking the components and joints. Each component and joint in the selection will have an orange outline.

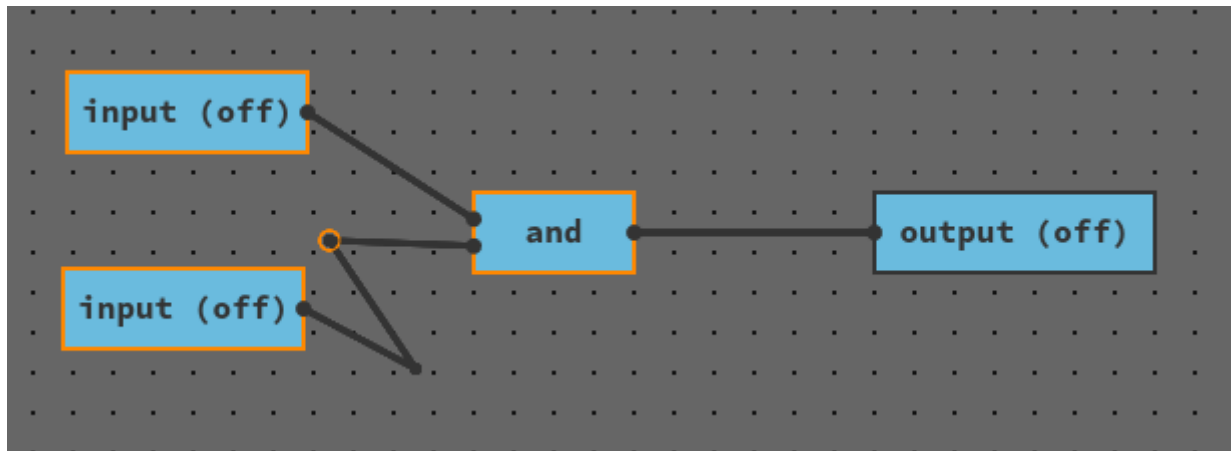


Figure 6: Multiple components and joints added to selection

Box selection can be used to select multiple items at the same time. By clicking and dragging starting on empty grid space, and covering each of the items you wish to select.

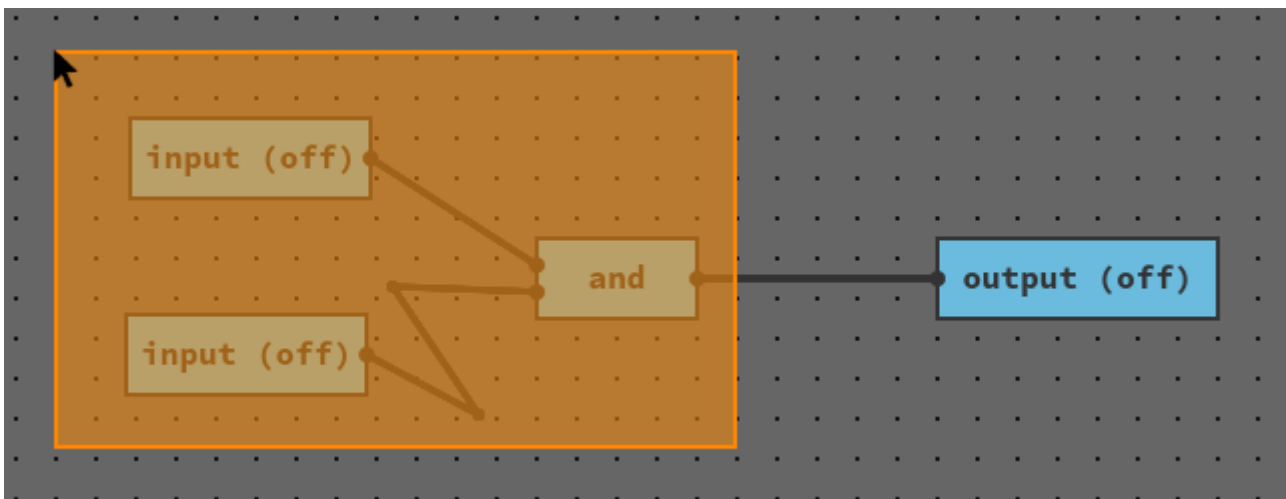


Figure 7: Selecting multiple items with box selection

Moving items

Components and joints can be moved. By first selecting the desired items, you can click and drag starting from one of the selected items. The items in the selection will move in accordance with your mouse. Releasing the mouse button will place the items in the new location. Connected wires will remain connected.

Deleting items

Items can be deleted from the circuit. First select the items you desire to delete, then press the 'Delete' key on your keyboard. The selected items will disappear from the circuit.

Activating inputs

Double-click 'input' components to toggle between *on* and *off* states. Toggling an input state will immediately be reflected in the circuit. Connected wires will activate and deactivate, and the signal will pass through the circuit.

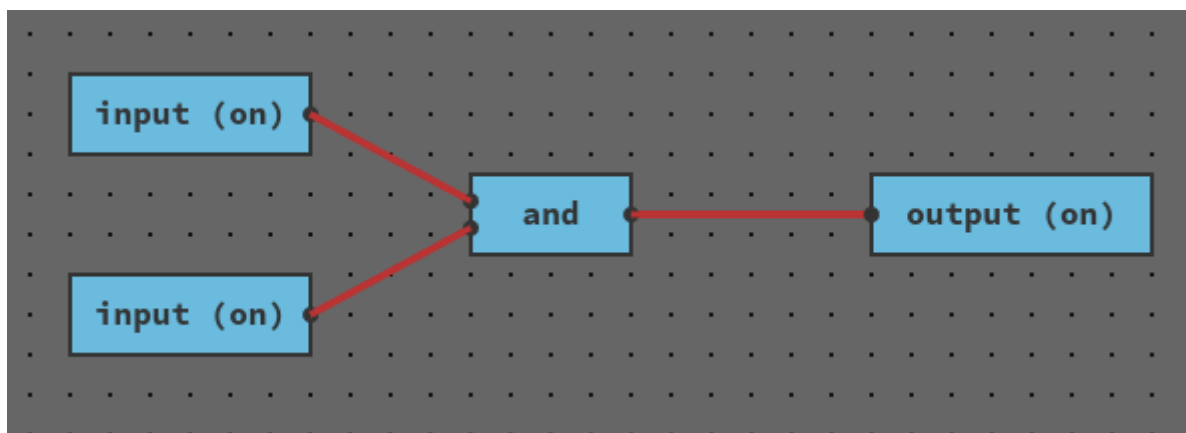


Figure 8: Circuit with inputs activated

To toggle activation of inputs, nothing must be selected. To make sure nothing is selected, it can be useful to click on empty grid area before double-clicking inputs, in case an initial double-click doesn't work.

Wiring

To wire components and joints together, click on a pin or a joint. When hovering the mouse over joint and pins, a white outline appears.

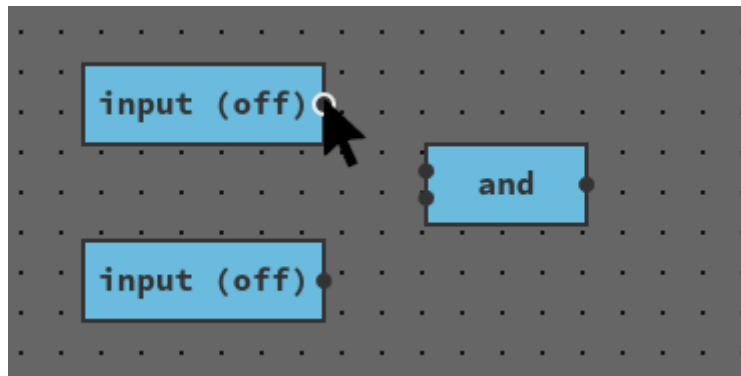


Figure 9: Hovering the mouse over a pin

Click to enter *wiring mode*. A wire will appear connected to the pin in one end and moving with your mouse on the other end. Clicking another pin or joint will connect those two together. When making the connection, you exit wiring mode. You can also exit wiring mode by pressing the 'Escape' key.

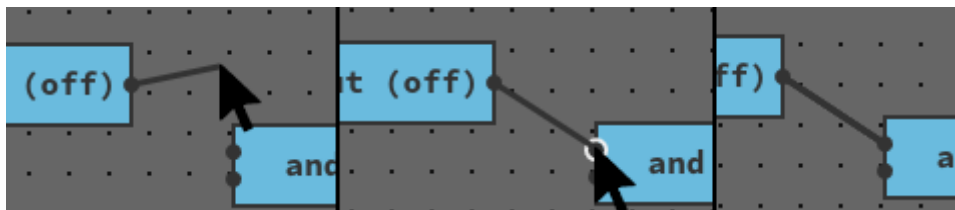


Figure 10: Two components being connected

In wiring mode, click empty grid area to make a joint. You will stay in wiring mode with a new wire connected to the new joint.

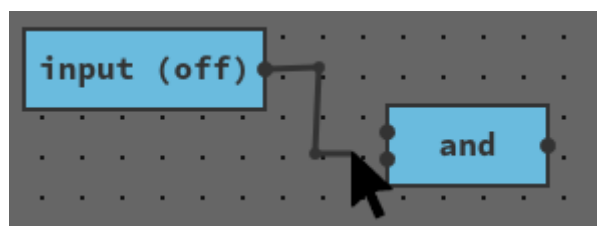


Figure 11: Joints in wiring mode

Toolbar

The toolbar is used to select components to place in the circuit editor. Initially the toolbar will list only the primitive built-in components. When saving components, they will appear in the toolbar.

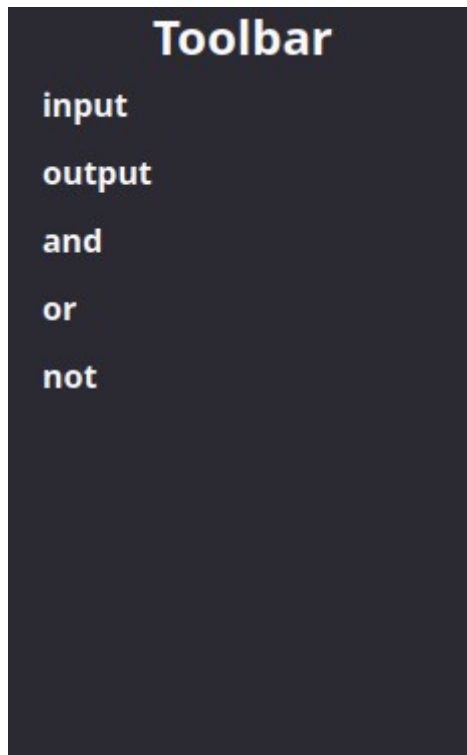


Figure 12: Toolbar initially

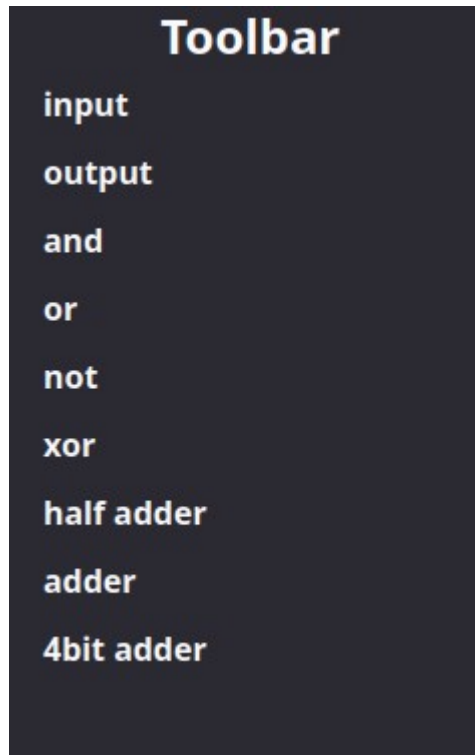


Figure 13: Toolbar with more components

Place components

To place a component in the circuit editor, select the desired component in the toolbar. The selected component will be shown with an orange underline.

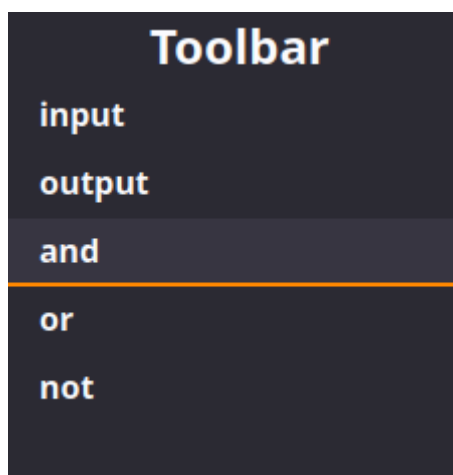


Figure 14: Toolbar with selected component

An outline of the component will now appear in the editor following the mouse.

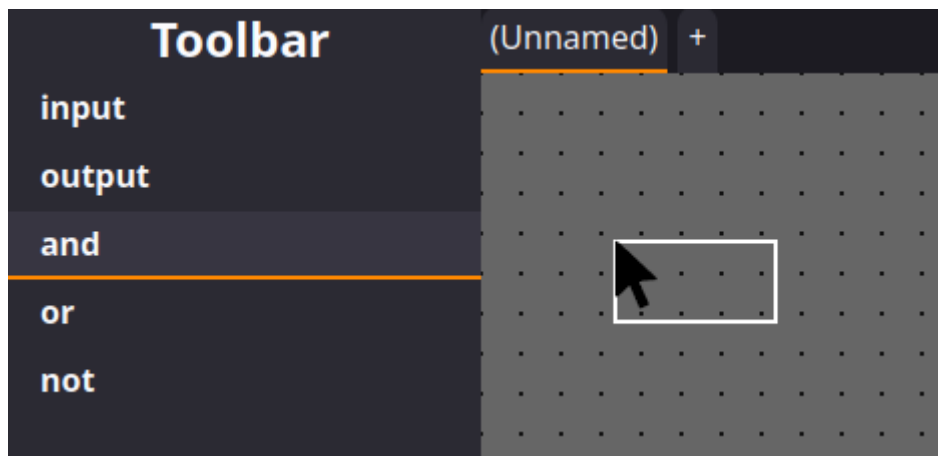


Figure 15: Outline shown in editor of component selected in toolbar

Clicking on empty grid area will place a component in place of the outline. To place multiple of the same component in succession hold the 'Shift' key while clicking.

Open component editor

For custom components, you can open their underlying circuit in an editor tab by double-clicking the component in the toolbar. If the component's tab is already open, it will switch you to tab. If it isn't, it will open it as a new tab.

Tab-bar

The tab-bar allows you to select between editors of different components. Each open editor will appear as a tab. Click a tab to switch to it. The currently selected tab is indicated with an orange underline.

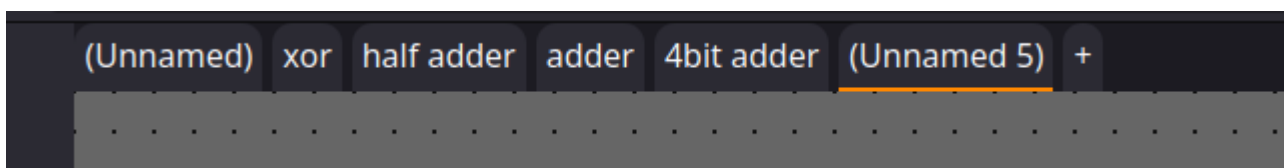


Figure 16: Tab-bar with multiple tabs

Create components

To create a new component, create a new tab by clicking the '+' button. Initially, the tab will have the name '(Unnamed 1)' or similar. You can use the component control buttons to rename and save the component.

Component control buttons

Use the component control buttons.

Rename component

When creating components, they will have the name '(Unnamed ...)'. To give them a name, click the 'Rename' button. A browser prompt will appear where you can input the new name.

Save component

To use a circuit as a component in other circuits, you need to *save* the circuit as a component. *After* giving your component an adequate name, click the 'Save' button. The circuit will now appear in the toolbar as a component. Notice that a component won't appear in its own editor tab; switch to another tab to see the effect.

Close editor tab

You can close the currently selected tab by clicking the 'Close' button. If the circuit is saved as a component, you can reopen it by double-clicking the component in the toolbar. Otherwise, the circuit will be discarded.

Example – An XOR component

To demonstrate the functionality of NandSim, explained here is an example of how to make a reusable XOR component.

An XOR gate has 2 inputs, one output, and can be described with the following truth table:

A	B	Result
F	F	F
T	F	T
F	T	T
T	T	F

Figure 17: XOR gate truth table

An XOR gate can be described using AND, OR and NOT gates. For the inputs A and B , XOR can be defined as $\neg(A \wedge B) \wedge (A \vee B)$.

To create this as a reusable component, open a new tab in NandSim.

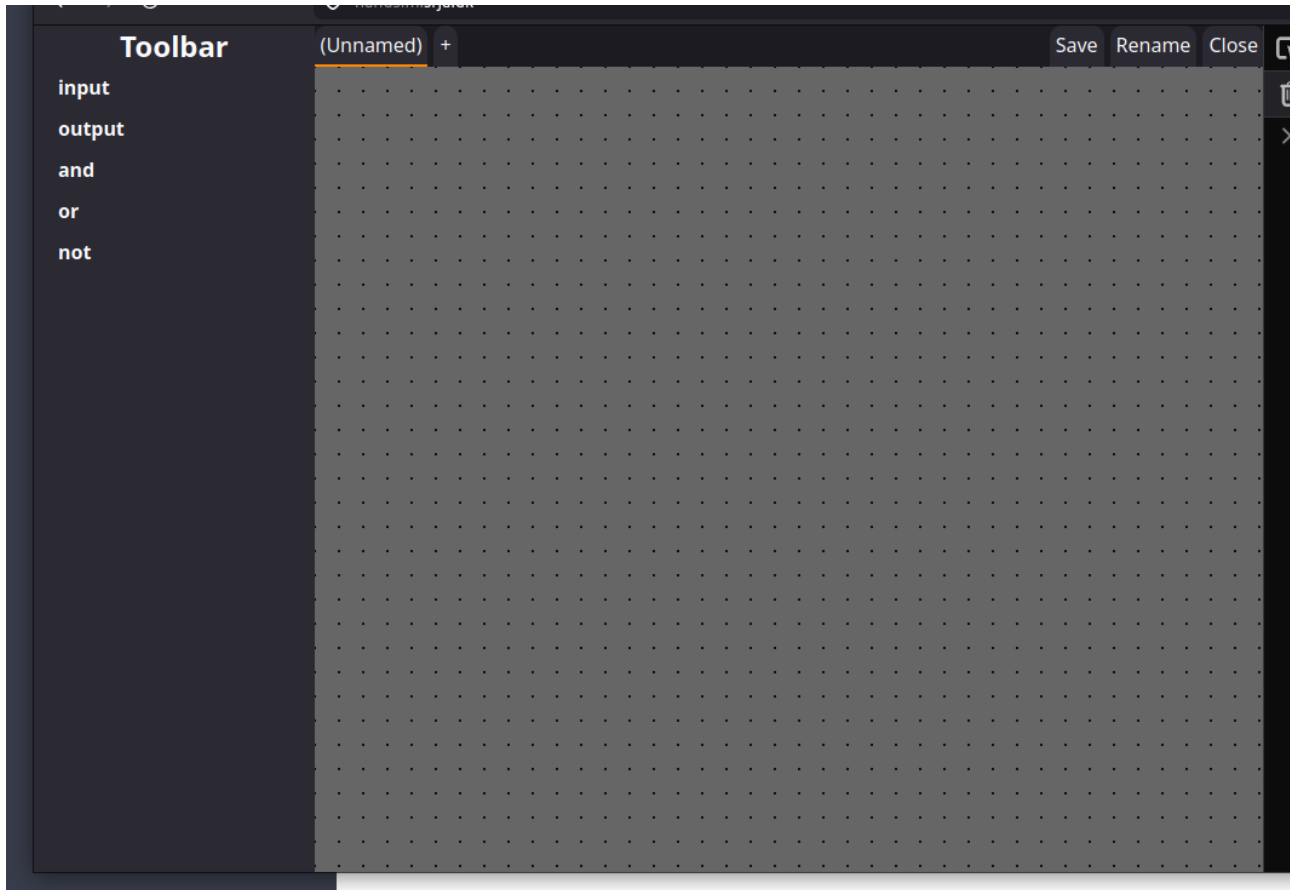
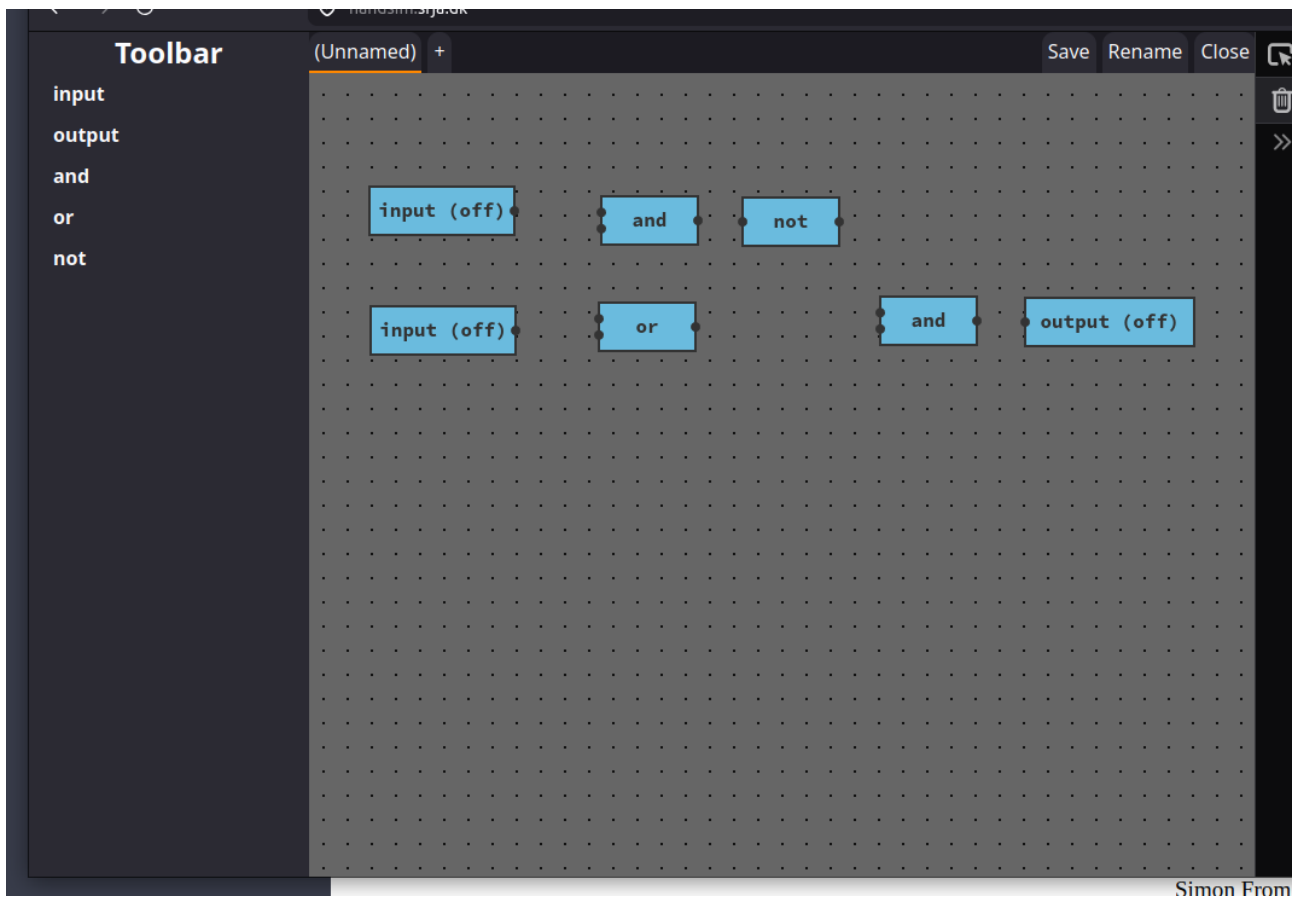


Figure 18: Empty tab

Next, place the components needed to satisfy the XOR behavior. Place 2 inputs, 2 ANDs, 1 OR, 1 NOT, and 1 output.



Simon From

Figure 19: XOR internal components placed

Next, wire together the components to complete the circuit.

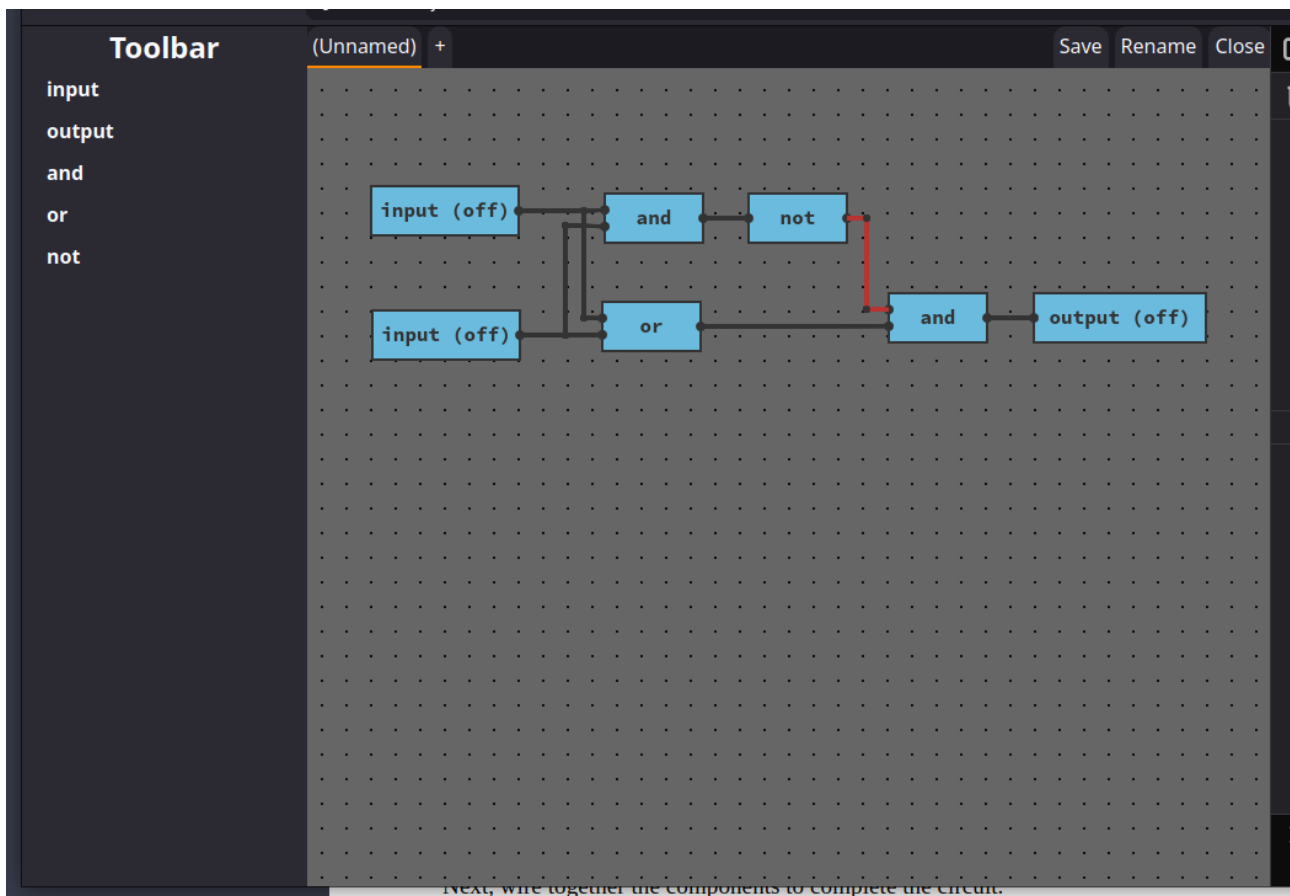


Figure 20: XOR components wired together

As can be seen, the output from the NOT gate is activated initially. After wiring the circuit test it out by trying different input configurations and examine the output.

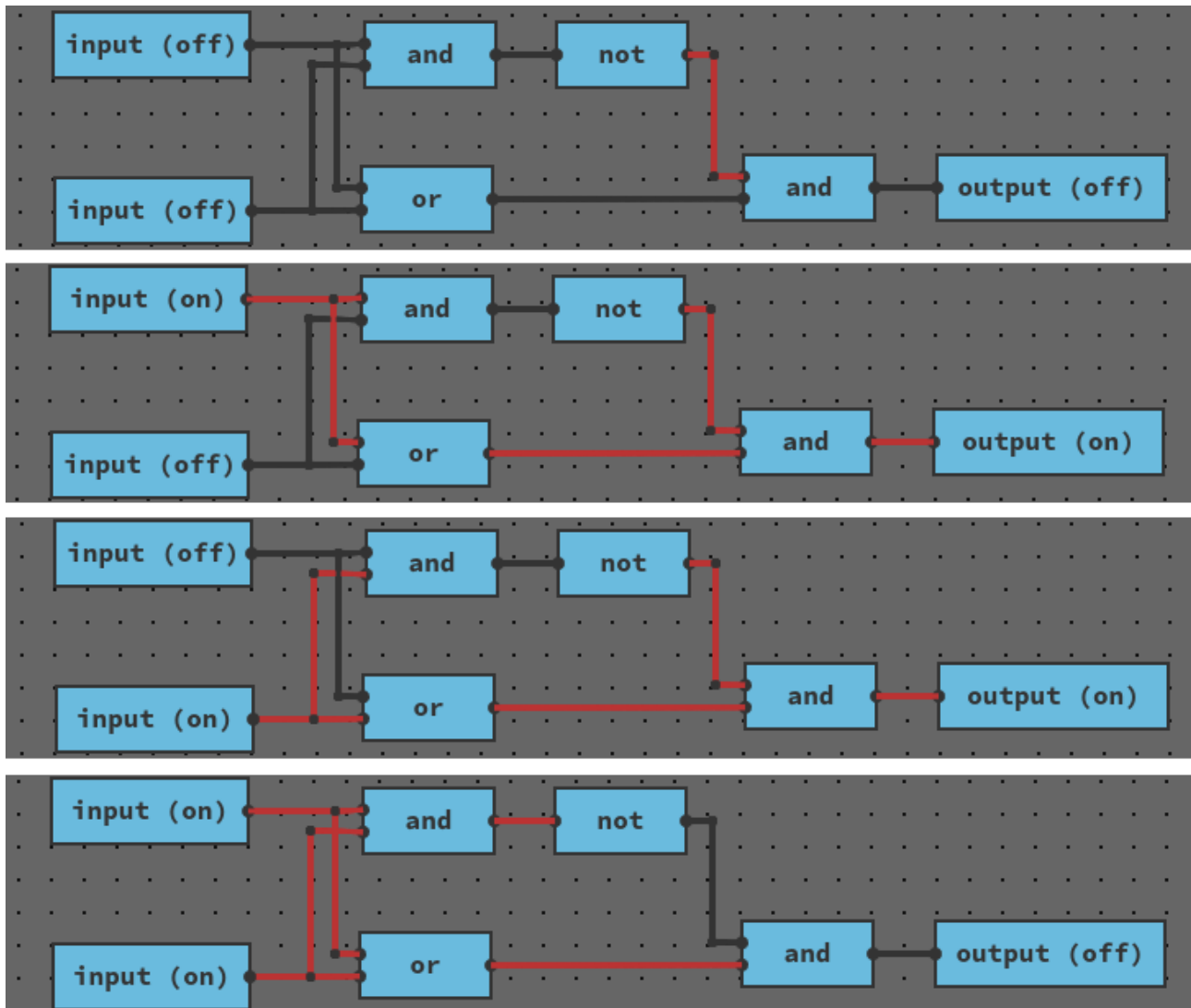


Figure 21: XOR all input combinations tested with corresponding output

Next, rename the component to “xor” with the ‘Rename’ button, and save it with the ‘Save’ button.



Figure 22: Save and Rename buttons highlighted

Your new XOR component is now created and ready to be used. Open a new tab to see the XOR component in the toolbar. Try using it in a new circuit.

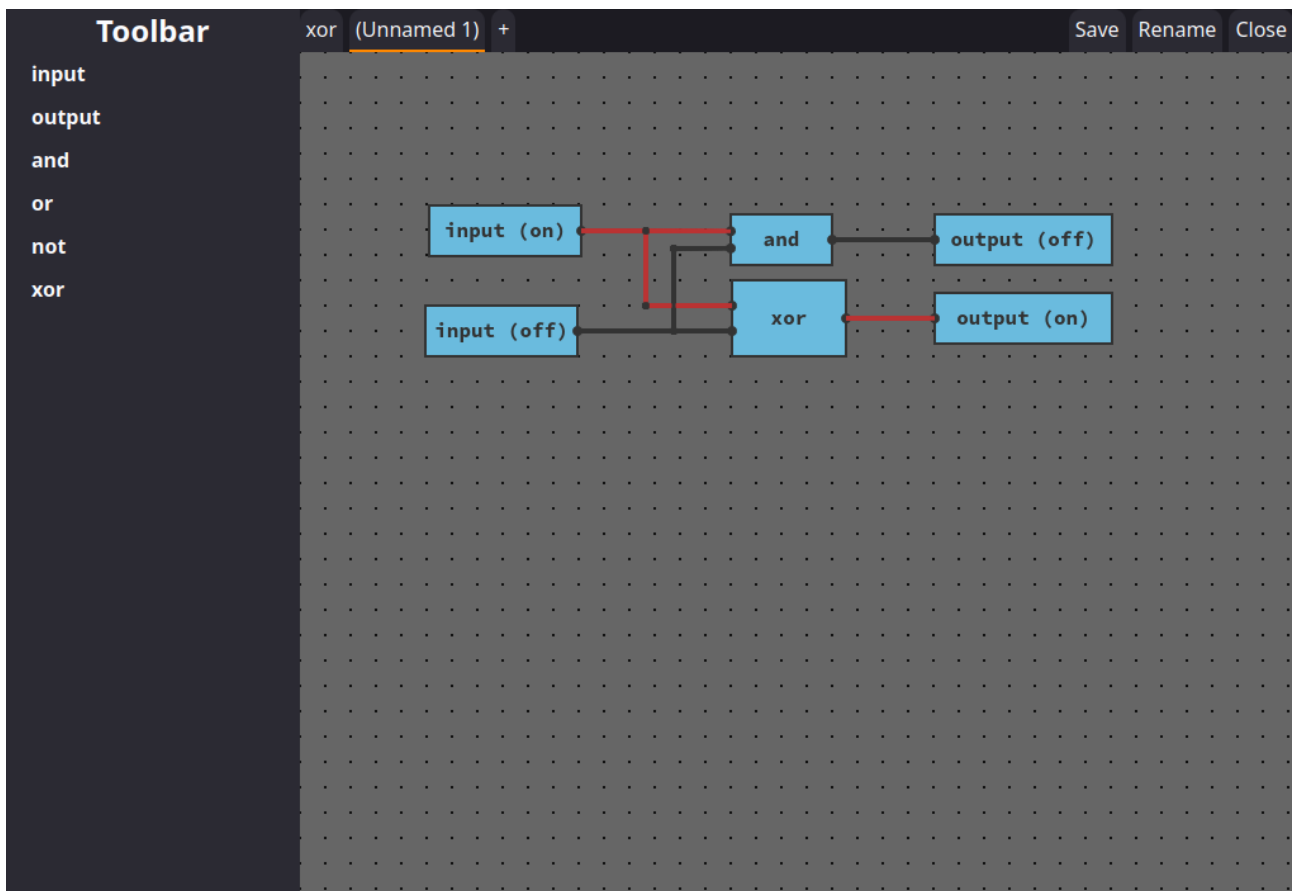


Figure 23: Circuit using XOR components

Product documentation

The purpose of this section is to detail how the application is implemented, and which decisions went into the development of it. This is technical documentation meant for developers seeking to contribute to the application.

The concept

As mentioned in the introduction, the objective is to decouple the development process of the firmware with the production process of the hardware. By decoupling firmware development, it can be initiated earlier in the process and be done in parallel with the time consuming hardware production phase, rather than sequentially afterwards. The approach to solve this problem is by introducing a software tool that the chip developers can use to build and simulate a digital twin of the hardware.

Digital twin

A digital twin is a simulated digital model of a product that has the purpose of emulating the behavior of the physical product. It is a digital counterpart that mimics specific behavior of the physical product. The desired behaviors are those that are relevant for the work needed to be done. In this case, the software tool should provide functionality to build a digital twin of an IC chip, that emulates the digital logic aspect of the circuit. The emulation should be mimic the behavior of the physical hardware in such a capacity that the digital twin can be used as a substitute for a physical hardware copy in the process of developing firmware.

The concept for the software tool is an application for building and simulating digital logic circuits. The application is a visual tool with a graphical editor, that IC developers, both software and hardware developers, can use to build and simulate digital logic circuits. The user can build circuits using components that represent inputs, outputs, logic gates, and previously defined circuits. The circuits can then be simulated interactively, so the user can observe the behavior of, and interact with built circuits.

Visual and interactive

Building circuits is a visual and interactive experience where the user places components on a board-area, resembling a printed circuit board or a breadboard. The user can wire up components in an ergonomic and intuitive manner, and visually observe the relationships and connections between the components of the circuits. The user is also able to edit parts of a circuits by either adding, moving, rewiring, or deleting components.

The user is able to simulate circuits interactively. When any change occurs to a circuit, such as a change in wiring or a change of the input states, the changed circuit is immediately simulated, and the resulting state reflected on the circuit visually. Both input and output components show their current state visually on the component itself. Wires change appearance depending on their activation state. This way, the user is able to quickly asses the input signals to a circuit, and trace the signals running through wires and components.

For circuits that depend on persistent state represented by cyclic connections, this state is preserved with a best-effort approach. If a circuit does not change, the state is guaranteed to persists. When a circuit does change, the simulator will preserve as much state as possible. For circuits whose state is non-deterministic on entry in theory is guaranteed to exhibit deterministic behavior in the simulator. This pertains for example to RS-latches that may be either low or high on entry, from the point of the definition. A constructed RS-latch will in the simulator always be one of the states initially.

Concretely, the application should be a graphical *desktop application*¹ that can be run on the user's own PC. The user should be able to use the PC's mouse and keyboard for input, and see visuals reflected real time on their monitor.

Satisfying requirements

To specify the details and functionality of the application, what follows is each requirement defined in the **Requirement specification** section followed by description of the functionality necessary to satisfy the requirement. Note that this is the design of the application, not the implementation.²

1 The term *desktop application* here means any application that can run on a PC, like a native desktop application in the traditional sense. The reason that this is important to note, is that the application is implemented as a web-application running in a web-browser on the user's PC. The application is designed as an application for use on a desktop PC. That it is hosted on a public URL-domain and runs in a web-browser is mere a detail of implementation.

2 Only a subset of the requirements, and designs to satisfy them, are included in the implementation.

Building circuits

★ *assemble logic circuits visually by placing and wiring together primitive logic gates.* This includes a visual editor area, where logic gates and components can be inserted and dragged around, and where each gate and component can be wired together.

The application shows a canvas-style element on the screen that will function as the application's *editor area*. The on the canvas is a rendered grid seen from a top-down view. The grid can be moved relative to the camera with the mouse. The grid extends indefinitely in all directions.

In addition to the editor area, the application also shows a toolbar. This toolbar is a list of all the components that the user can use in a circuit. Each entry in the list is a button, that the user can click, which will subsequently allow the user to place that type of component as part of a circuit in the editor area.

Placed components can be selected, moved and deleted using mouse and keyboard inputs.

Components can be wired together by connecting visually apparent connection points (pins) with the mouse.

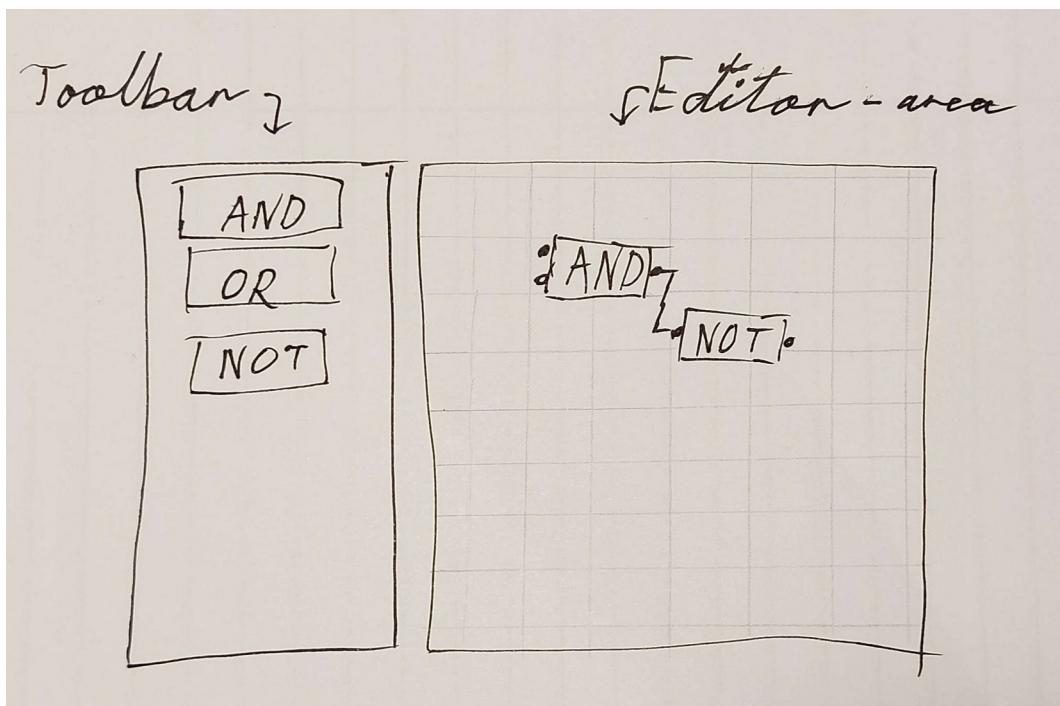


Figure 24: Sketch of toolbar and editor-area

Initially, the toolbar consists of primitive logic gates in addition to the ‘input’ and ‘output’ components.

Thus the user can assemble logic circuits visually. The user can select components from the toolbar, either input and output components or components representing primitive logic gates, and place these components in the editor area as part of the circuit. The user can then visually wire together the placed components, finishing the circuit, and thus the requirement is satisfied.

Create components

★ create circuit components built with logic gates that can be used like primitive logic gates. *To help organize and make complex circuits manageable, sub-sections of a circuit can be encapsulated in a component.*

In addition to the editor-area and the toolbar, the application shows a tab selection bar (tab-bar). This tab-bar is used to switch between different components. Each tab contains an editor of a distinct component with an associated name.

The circuit in one tab can be saved as a component and used as a regular component in other circuits (other tabs). A saved component where the editor tab has been closed can later be edited again, where an editor tab will be opened with the original circuit. The component can then be saved, overwriting the older version. A renamed version can also be saved, thus preserving the old version with it’s original name, and the new version with the new name.

The amount of inputs and outputs of the circuit is reflected on the component. Every ‘input’ component becomes an input-pin on the component, and each ‘output’ component becomes an output-pin. Connecting an input-pin on the component to an active signal is equivalent to activating the associated ‘input’ component manually. The order of the input- and output-pins is controlled by the vertical position of the ‘input’ and ‘output’ components, meaning for example that the topmost placed ‘output’ component in the circuit will correspond to the topmost output-pin on the component. All inputs and outputs appear on the component regardless of if they’re connected or not.

Thus by offering functionality to edit different circuits in separate editors, saving them as components, and allowing the user to use the saved components in other circuits, the application satisfies the requirement.

Simulate circuit

★ ***simulate the logic of a circuit.*** A simulation consists of initializing a state according to the circuit and the inputs given. The simulation should be continuous, and support simulating state dependent on a previous state.

★ ***interact visually with a simulation of a circuit.*** The simulation should support updating the state according to any changes made to the circuit.

As explained above, the application simulates the circuit continuously. This is reflected by the wires changing appearance in accordance with the signal that passes through. The simulation runs on demand, meaning everytime a change happens that could affect the state of the circuit, a simulation run executes. As also explained above, the states of circuits are preserved in a best effort approach, so that the simulation of circuits depending on state will depend on the state of the previous simulation run. Thereby, these requirements are also satisfied.

Upload program and data files

Upload files with data input for complex circuits, including program and data files in suitable file formats such as binary or hexadecimal. The purpose is to enable flashing of a simulated circuit with data and programs to be used in the simulation.

The application, in addition to the aforementioned built-in components, also contain components that emulate physical ROM components. These ROM components stores N-bit data values that can be addressed using N-bit address values. ROM components can be added with a specified amount of address bits and value bits.

Initially, the values stored in a ROM component are all zeros. By double clicking a ROM component, a memory-editor pop-up window appears. In this window, the value of each address can be inspected and changed.

Additionally, there is a button to upload a data-file. Data-files contain values to be loaded into the ROM component. Data-files are text files with a specific format. The format is white-space-separated integers, represented by base-16 hexadecimal. Each integer a value, and the order of the integers are the same as the order of addresses. The values are truncated to fit as the N-bit data values. With this, the requirement is satisfied.

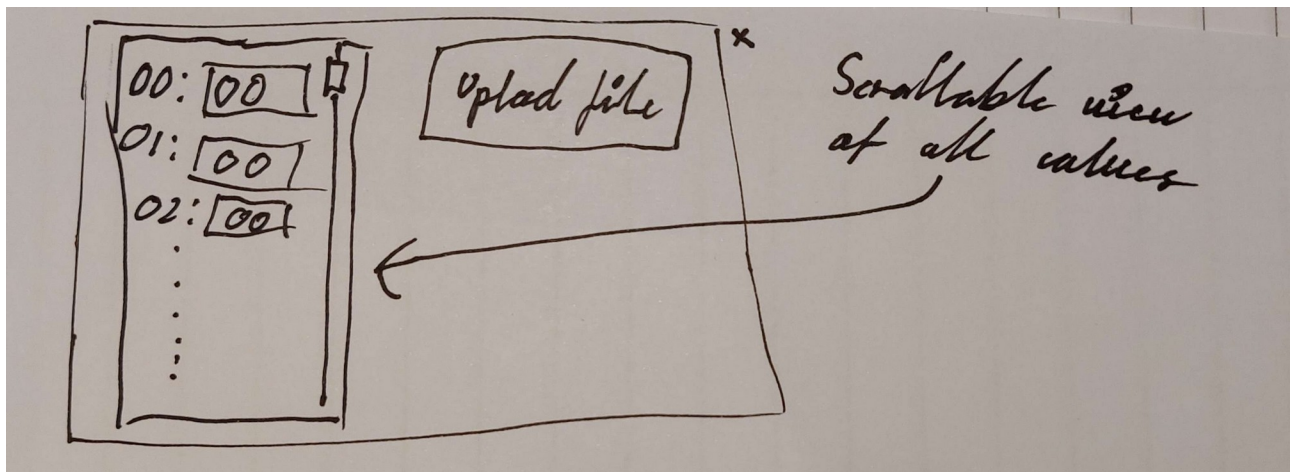


Figure 25: ROM editor pop-up window

Solution (Problembeskrivelser)

The following section describes the implementation as a set of problems that I faced during the development process, and the solutions to these problems. These problems have been identified and solved during development, and the solution is explained here, as well as which decisions went into said solutions.

Graphical desktop application

The application is developed as a graphical desktop application, that incorporates visuals, real-time updates, and user input controls. The major challenge in this regard is selecting a set of technologies both are able to satisfy the requirements, and that also enable the implementation of the designed application. This decision has a large impact of the project, as a major part of the overall implementation is dependent on the technologies chosen for the graphical part of the application. Part of the architecture develops out of the choice of technologies.

I chose to implement the application as a desktop web-application, meaning an application that runs inside a web-browser such as Edge and Firefox. Running inside a web-browser limits the options in some aspects for technology choices, but opens for other possibilities, such as having access to the JavaScript³ ecosystem. I decided to use React⁴ as a frontend web interface library. And further, I decided to use TypeScript⁵ as the main programming language. And lastly for the application, I decided to use Node.js⁶ and Vite⁷ as development tooling.

The decision to make the application web based has two reasons. The first reason is that my prior experience in web technologies vastly outweighs my experience using other technologies for desktop development. I assumed this to be important, as I estimated that the application obtain significant complexity. To be able to satisfy the requirements within the given time frame, I assumed I needed to approach the problem with pre-existing knowledge of how to manage this complexity. The assumption was correct.

The second reason is that it opens up for the JavaScript/web ecosystem. To my estimation, this ecosystem is one of the most expansive and supported ecosystems. In unforeseeable events where I would have to pivot on the development approach involving different choices of dependencies, an expansive and support ecosystem will be the most advantages. Using technologies with more limited ecosystems, you have less options regarding choices of dependencies and tooling.

The decision to use React, TypeScript, Node.js and Vite, followed from the previous decision. These are the technologies that I am the most confident in using to solve problems similar to application. Vite is to my estimation the industry standard for bundling web applications, something that's required when deploying applications with dependencies to browser environments. Vite is best supported Node.js as a JavaScript runtime to run locally on the development machine.

The only common alternative to using TypeScript is using JavaScript. Off the name, the difference between JavaScript and TypeScript can deduced. I decided to use TypeScript, as I from experience know that the type system that TypeScript provides is a very great advantages when designing and implementing complex applications. I utilized several rather advanced patterns during development that relies heavily on the expressive type system.

3 MDN: JavaScript, <https://developer.mozilla.org/en-US/docs/Web/JavaScript>, visited June 2026

4 React: React, <https://react.dev/>, visited June 2026

5 Typescript: TypeScript is JavaScript with syntax for types., <https://www.typescriptlang.org/>, visited June 2026

6 Node.js: Run JavaScript Everywhere, <https://nodejs.org/en>, visited June 2026

7 Vite: The Build Tool for the Web, <https://vite.dev/>, visited June 2026

As an alternative to React, I considered implementing the application without an external user interface library. In that case, I would be using web standard provided APIs directly to manipulate the *DOM*⁸ of the browser. This is largely comparable in complexity to using a library like React. It solves the same problem. The libraries remove complexity in some aspects, but add complexity in other aspects. When using the the web APIs, my experience is that the largest problem is designing the implementation in a composable way. React, at it's core, is about providing a mechanism to design interfaces in a composable way. The complexity, when using React, is instead placed in managing state-changes and re-renders.

For the decision to implement the application as a web application, I considered some alternatives. One alternative I considered was implementing the application using a cross platform application framework like Dear ImGui⁹ for C++. I have extensive experience writing software in C++, and I know Dear ImGui to be sufficiently capable as a GUI library. The concern I had was that the application needs a canvas-like element with custom drawing. I was not confident that I could implement custom drawing in Dear ImGui sufficiently in the given time frame. Another alternative could have been Flutter or Maui, or any native GUI toolkit. Given my lack of experience with any of those, I decided against it.

One problem of web applications are than the experience of using them can be that of web sites rather than desktop applications. A solutions to this is to run the application in browser that appears as a desktop app running natively, with a natively running runtime on the desktop PC. The Electron¹⁰ toolkit serves this exact purpose. There are other options too for this, but Electron is by my estimation most popular by far. The approach of running a web application as though it's a desktop application is rather common, with examples including VS Code and Discord.¹¹

Using Electron would have given me another advantage, that I included in consideration. The advantage would be ability to run software natively on the user's machine. When running as a web application, the access to the user's computer is limited. The web standard provides only a limited interface to the user's computer. This was especially a consideration in relation to my considerations

8 MDN: *Document Object Model (DOM)*, https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model, visited June 2026

9 Github: *Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies*, <https://github.com/ocornut/imgui>, visited June 2026

10 Electron: *Build cross-platform desktop apps with JavaScript, HTML, and CSS*, <https://www.electronjs.org/>, visited June 2026

11 Ibid.

about how performant the simulation needed to be. I decided during development that the browser's facilities regarding performance are sufficient, and that it would take too much time to implement the simulator outside the editor. The decision to implement the application as a desktop application running in the browser was therefore a correct decision.

Events

The application is a graphical desktop application with complex user interaction. The application needs to be able to respond to a wide array of events that can occur at any point in the run time of the application. To manage this complex environment, and at the same time allow loose coupling and cohesion in the application, the application is implemented with an *event-driven architecture*.

*User interaction uses both the keyboard and the mouse, and the next event might come from either source. [...] Programs need to be written in a way that can cope with an unpredictable sequence of events from several sources, and this approach is generally referred to as **event-driven** programming.*¹²

The event-driven architecture contains at its heart the **EventBus** class.¹³ An instance of the **EventBus** implements functionality for sending events and subscribing to certain events. Large parts of the application shares an aggregation relationship of the class to this instance. The instance is compositionally defined in the **Editor** class.¹⁴

Many parts of the program, those that have to react to user input or other seemingly suddenly arising occurrences, are implemented as *observers*.

*The key objects in this this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. [...]*

*This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.*¹⁵

¹² Introduction to Operating Systems, pp. 34-5

¹³ Defined in `app/src/editor/events.ts`

¹⁴ Defined in `app/src/editor/Editor.ts`

¹⁵ Design Patterns – Elements of Reusable Object-Oriented Software, p. 294

An example of a component implemented as an observer of events in this regard is the **ViewPos** class.¹⁶ The **ViewPos** class is responsible for keeping track of the camera position relative to the editor-area. It's concrete responsibilities are keeping track of the *offset* position, and relaying mouse events corrugated according to the offset position.

ViewPos takes as a parameter to it's constructor a reference to the event bus. In the constructor, the function **EventPos.subscribe(...)** is called with a set of event tags and a callback function. This sets up the subscribe mechanism.

```
export class ViewPos {  
  ...  
  constructor(private events: EventBus) {  
    this.events.subscribe(  
      ["MouseDown", "MouseMove", "MouseClick", "MouseDoubleClick"],  
      (ev) => {  
        ...  
        switch (ev.tag) {  
          case "MouseDown":  
            ...  
          case "MouseMove":  
            ...  
          case "MouseClick":  
            ...  
          case "MouseDoubleClick":  
            ...  
        }  
      },  
    );  
  }  
  ...  
}
```

According to the *Interpreter pattern*,¹⁷ switching on the event tag, the event handler decodes the event and runs some associated behavior accordingly.

The **ViewPos** class also sends events. When the event handler receives a *MouseDown* event, the class will submit a new *MouseDownOffset* event, that sends the mouse position according to the current offset.

```
    case "MouseMove":  
      this.events.send({  
        tag: "MouseMoveOffset",  
        pos,  
        deltaPos: ev.deltaPos,
```

¹⁶ Defined in *app/src/editor/ViewPos.ts*

¹⁷ *Design Patterns*, p. 243

```
});
```

The **ViewPos** therefore also acts as a *subject*.

The advantage of **ViewPos** using events is that the class is completely spatially decoupled from both the UI code sending the *MouseDown* event, and the consumers of the *MouseDownOffset*.

*In event-based architectures, processes essentially communicate through the propagation of events, which optionally carry data, [...] event propagation has generally been associated with what are known as **publish/subscribe systems** [...]. The main advantage of event-based systems is that processes are loosely coupled.¹⁸*

The reason for the decoupling using this approach, is that the publisher does not care about the subscribers when sending a message. It just sends a message, and the consumers are themselves responsible for subscribing to those messages.

We don't know (or care) what happens to the message after we have sent it. We have done our job and told somebody else that something has happened.¹⁹

There are many kinds of events defined in the system. Not all are directly related to user input. Some events are sent as responses to receiving and handling user input. Switching tabs for example results in a *SimulateRequest* being sent, so that the circuit on the switched-to tab will have an up-to-date state. Events in this sense are just used when one part of the system communicates with some other part of the system that might want to react to something.

An event is just something that happens—something noteworthy that the programmer thinks somebody should do something about.²⁰

The events are defined centrally in the **Event** type.²¹ Some events contain data, others are empty and merely represent the notification. The type is defined using TypeScript's union type, so that the **tag** field can be used to distinguish different kinds of events, and to access the associated data in a type safe manner.

18 *Distributed Systems*, p. 35

19 *Programming Erlang*, pp. 330-1

20 *Programming Erlang*, pp. 330-1

21 Defined in `app/src/editor/events.ts`


```
export type Event =  
  | { tag: "MouseDown" | "MouseUp"; pos: V2 }  
  | { tag: "MouseMove"; pos: V2; deltaPos: V2 }  
  | { tag: "MouseLeave" }  
  | { tag: "MouseClicked" | "MouseDoubleClick"; pos: V2 }  
  | {  
    tag:  
      | "MouseDownBegin"  
      | "MouseDown"  
      | "MouseDownDragBegin"  
      | "MouseDownDrag";  
    pos: V2;  
    deltaPos: V2;  
  }  
  | { tag: "KeyDown" | "KeyUp"; key: string }  
  | { tag: "SelectTool" | "ShowSelectedTool" | "OpenTabWithTool"; tool: string }  
  | { tag: "CreateTab" }  
  | { tag: "SelectTab" | "ShowSelectedTab"; idx: number }  
  | { tag: "MouseDownOffset"; pos: V2; absPos: V2 }  
  | { tag: "MouseMoveOffset"; pos: V2; deltaPos: V2 }  
  | { tag: "MouseClickedOffset" | "MouseDoubleClickOffset"; pos: V2; absPos: V2 }  
  | { tag: "RenderRequest" | "SimulateRequest" | "SaveRequest" }  
  | { tag: "SaveComponent" | "CloseComponent" }  
  | { tag: "RenameComponent"; newName: string };
```

When subscribing to events, you specify the events you wish to receive. The event bus will only send those kinds of events to the event handler.

*The basic idea is that processes publish events after which the middleware ensures that only those processes that subscribed to those events will receive them.*²²

This is primarily a concern of performance, but it helps with debugging too. For example, that subscribers have to state what events they want to receive explicitly, makes it easy to find all event handlers for a specific event.

The event bus is implemented with a two-way map approach using **Map** instances. This approach ensures theoretically $O(\log N)$, practically instant, lookup for subscriber handlers. A two-way map is needed for this, because a subscriber has to be able to unsubscribe their handler. To do both the lookup and the unsubscribing fast, you need both the mapping from subscriber actions to their corresponding event tags, and from event tags to their corresponding subscriber actions.

²² Distributed Systems, p. 35

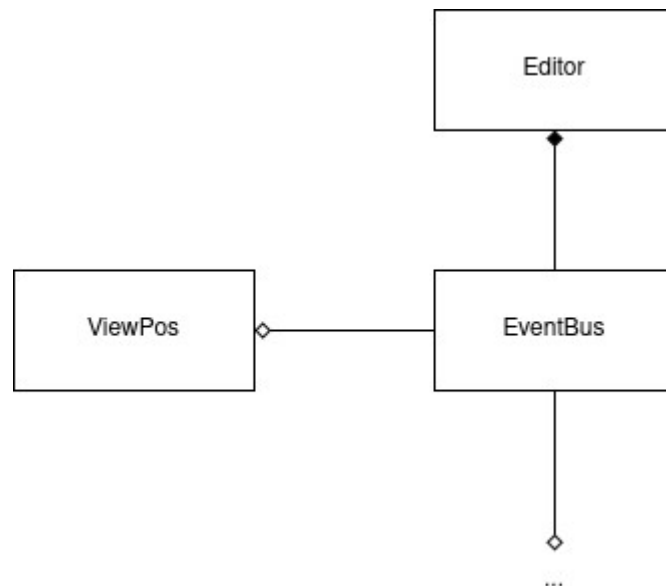


Figure 26: High-level class diagram showing the relation between Editor, EventBus and ViewPos

Editor

The **Editor**²³ class is the central component of the application. An instance of this class is instantiated when the application starts, and this instance lives throughout the entire duration of the program, meaning when the browser tab is reloaded or closed.

The **Editor** class delegates many responsibilities to subclasses according to the *Composite pattern*.²⁴ Communication with subclasses is done with two approaches. Some communication is done using direct method calls. Other communication is done using the shared event bus. Communication using events is predominantly used when tight coupling of the receivers to the sender is undesirable. Direct method calls are preferred, if there is a direct relationship between the sender and the receiver.

The **Editor** class's primary responsibility is maintaining the *state* of the editor in the application. The state is here similar to mode. When using the editor, each of the different behaviors is

²³ Defined in `app/src/editor/Editor.ts`

²⁴ *Design Patterns*, p. 163

represented by a distinct mode. The editor uses the *State pattern*²⁵ to extract the variant behaviors into each their distinct state-class.

*Most popular interactive drawing programs provide “tools” for performing operations by direct manipulation. For example, a line-drawing tool lets the user click and drag to create a new line. A selection tool lets the user select shapes. There’s usually a palette of such tools to choose from. The user thinks of this activity as picking up a tool and wielding it, but in reality the editor’s behavior changes with the current tool: When a drawing tool is active we create shapes; when the selection tool is active we select shapes; and so forth. We can use the State pattern to change the editor’s behavior depending on the current tool.*²⁶

The editor being developed in this project, just as in the example explained above, supports multiple different kinds of functionality. Depending on what ‘tool’ the user has selected. The editor’s behavior will reflect that. For example, when pressing the ‘Shift’ key and dragging with the mouse, the editor will switch to **Panning-state**.²⁷²⁸ The **Editor** class functions as the *Context* in the state pattern.

The advantage of this approach is that each state or mode of the editor is encapsulated as a separate State-class. Adding, removing, or changing states is a trivial endeavor. Combined with the event-driven architecture, this approach has shown itself to be advantageous.

The State pattern is implemented by defining an interface **State**. The **Editor** class stores an instance of this interface. Each state is implemented as a class implementing **State**. Initially, **Editor** constructs a **Normal** state.

The state classes communicate with the **Editor** class through public methods and public fields. One disadvantage of this pattern is that the *Context* class has to expose its implementation details, so that the state classes can access them. This has been a very little problem in practice.

The state transitions is defined both in the states themselves, but also in the **Editor** class. The state interface define the methods, **enter()** and **leave()**. These get called on each state transition. The

25 Ibid., p. 305

26 *Design Patterns*, p. 313

27 Defined in app/src/editor/Editor.ts

28 Note that Panning is the incorrect term, as it means angular adjustment. What is meant is movement of the camera relative to the editor area.

leave-function is especially important, as each has to unsubscribe from the event bus, when the state is transitioned away from. Since JavaScript is a garbage collected language, there is no mechanism to automatically clean up residue (event handlers for example) from objects cease to be used. In this case, the residue will keep the class from being garbage collected, and worse, the event handler will still be active and listening for events. It's for that reason imperative that transient object that subscribe to events also unsubscribe.

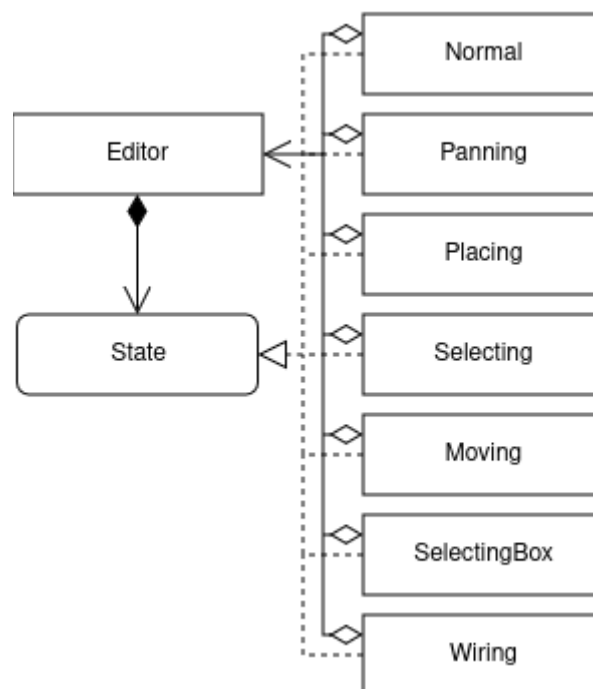


Figure 27: High-level class diagram showing the State pattern

Project

The **Editor** class delegates the responsibility of keeping track all the things related to a specific project. This is delegated to the **Project** class.²⁹ This class is responsible for keeping track of all of the data, or rather non-transient state, in a given project. A project includes all saved components, their circuits, and all open editors. The **Project** class is responsible for supporting the functionality of switching between open editors, and providing information about the project, such as which components are available.

²⁹ Defined in `app/src/editor/Project.ts`

The **Project** class contains an instance of the **ComponentRepo** class and a list of instances of the **Board** class. Instances of the **Board** class represent in-editor circuits. The **ComponentRepo** class³⁰ stores every component-kind of the project. When a circuit is saved as a component, it is added to the component repository. The component repository provides the underlying functionality of the toolbar. The component repository also stores a serialized version of the underlying circuit for each component. This enables the functionality of re-opening the editor for a component.

The **Project** class is also responsible for saving and restoring everything related to the project. This involves serializing and de-serializing the data of the project, and storing and retrieving the data from the browser's local-storage.

Serialization and de-serialization is implemented in an object-oriented manner, where the logic of serializing and de-serializing each object is delegated to the object itself. Serialization results in a serializable data structure, that is, a purely hierarchy of JavaScript object and arrays that can be converted to JSON and back with no loss of information. Objects that refer to the same shared instance objects will, when serialized, refer to those shared objects with indices into central arrays of all the shared objects. De-serialization re-constructs all of the instances from the serialized data-structure. After serializing a project and de-serializing it again, all the instances are re-instantiated but with the relationships from before serialization preserved. When saving the project, the serialized data structure gets stored as a JSON string in the browser's local-storage using the web standard's **LocalStorage** API. The entry is saved with the name "nandsim".

The **Project** class is instantiated with its static **Project.loadLocalStoreOrInitNew()** method. This method checks if the local-storage contains an entry called "nandsim". If such an entry exists, it will instantiate a **Project** instance by de-serializing the stored data. If no such entry exists, a new **Project** will be instantiated with default values.

One of the non-functional requirements is:

***Preventing loss of work.** The system should employ a strategy to save the user's progress frequently enough, so that work is never lost. Immediately following any significant change, the system should be able to restart without loss of work.*

30 Defined in app/src/editors/board.ts

Every time a change is made to the **Project** instance or any of its sub-instances, a *SaveRequest* event is published. This event will cause the **Project** to save itself. This means that after every significant change, the project is saved to local-storage. And when it is saved in local-storage, it can be retrieved when the browser window is refreshed or closed and re-opened. Thus the project fulfills this requirement.

An alternative to storing saved projects in local-storage is to store them on a server. The requirement specification specifies that projects can be stored in a cloud solution. I made the decision to only store projects locally, to prevent having to implement the cloud solution. This was a decision taken on the basis of time constraints.

Board

Instances of the **Board** class³¹ represents in-editor circuits. This class is responsible for everything pertaining to the ‘circuit board’ (hence the name). When the **Editor** class manipulates a circuit, it manipulates it using an instance of this class. This class is responsible for storing, serializing, de-serializing circuits, is responsible for knowing the positions of placed items, and it is responsible for simulating and rendering the circuit visually. These responsibilities are in large part delegated to respective sub-components.

Building circuits is done using the **placeComponent()**, **addJoint()**, and **addWire()** methods. These methods are called from the state-instances of the **Editor** class. Circuits are represented by objects of the **Component**, **Wire** and **Joint** types.³² Connections between these instances are represented by these objects referring directly to each other. Concretely, it is the wire-objects that has aggregation relations to components and joints. The **addWire()** method takes as input the object themselves that the wire is supposed to connect. The way that the **Editor** knows which components and joints to pass to **addWire()** is by using the **handleMouseClicked()** method. This method takes a mouse position and a set of callback functions for specific cases. These cases include if the mouse is above a component, a input pin, an output pin, a joint, or a wire. In these cases, the associated callback function is called with the item passed as a parameter. The responsibilities are therefore separated such that **Board** is responsible for knowing the positions of items, and at the same time, **Editor** is able to retrieve instances of clicked items.

³¹ Defined in *app/src/editor/Board.ts*

³² Defined in *ibid.*

The **Board** class is responsible for rendering itself, meaning the circuit of places components with it's joints and wires. Additionally, the states of components should be shown visually, as well as the signals passing through wires according to the simulation. Because rendering occurs as an independent *process*³³, an instance of **Board** has to store the state needed for rendering internally. This includes for example storing a set of wires that are activated, and storing if an item is being hovered by the mouse. The drawing itself is delegated to the **Renderer** class. In **Board's** **render()** method, an instance of the **Renderer** class is passed.

Lastly, the **Board** class is responsible for simulating the built circuit. This involves producing an internal representation (IR) of the circuit appropriate for simulation, simulating the circuit based on the IR, and then updating the internal state in **Board** according to the result of the simulation. The simulation is delegated to the **Sim** class. That as well as the building up of the IR, I will explain in the next section on **Simulation**. Updating the state based on the simulation is done by keeping track of the IR's and **Sim's** state-objects and output results. The **Board** class keeps a mapping between produced IR and the **Board's** own items. This approach allows **Board** to relate the results of simulation to the circuit. The implementation is defined in the **simulate()** and **toIr()** methods of **Board**.

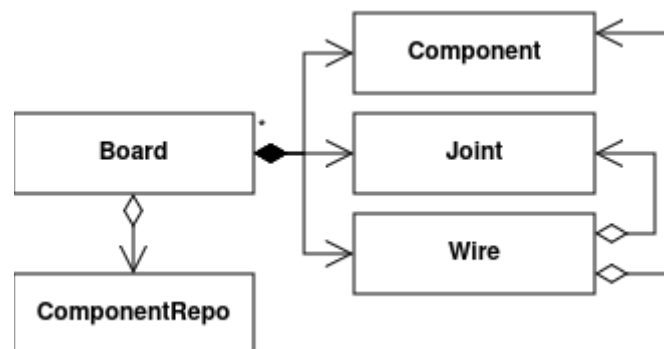


Figure 28: Class diagram of Board

Simulation

The simulation is implemented with two conceptual concepts. One is the *internal representation* (IR^{34}), the other is the simulation logic encapsulated in the **Sim** class. The goal of simulation is to

³³ Process is here meant conceptual process, not hardware thread or operating system process.

³⁴ Conventionally, IR refers to *intermediary* representation. My original intention was to transform the IR to another representation for simulation. I decided later on to simulate the IR directly. I therefore changed the acronym to mean *internal* representation.

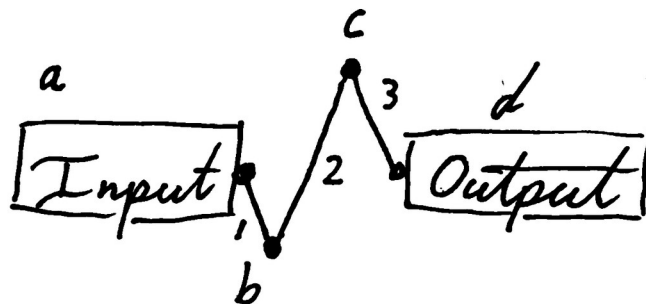
calculate the state of a circuit depending on its input and previous state, in a correct and efficient manner.

Simulation is conceptionally split up into separate steps. The first step is building the circuit IR. This is done using a lowering algorithm in **Board**. The second step is optimizing the IR. The purpose of this is to make the IR more efficient to simulate while preserving the behavior. This step is desirable, as the lowering algorithm uses a ‘dumb’ approach of generating IR from the circuit. The third step is the simulation step. Here the circuit IR is simulated using the simulation algorithm implemented in the **Sim** class. The fourth and last step is handling the result of simulation. This last step was described in the **Board** section.

IR

To represent circuits for simulation, a distinct representation is used than the one used for building the circuit. The reason for this is that the circuit in editing, and the circuit in simulation has different desirable traits and characteristics. The crucial difference is in the information that the representation represents.

The in-editing representation, the one used in **Board**, is appropriate for circuit construction. The representation represents the positional relations between items, and the mechanical wire connections between the components and joints. The components and joints exist independently. It is only the wires that know which components and joints they are connected to in each end. This is advantages for editing circuits visually, as it gives certain liberties for invalid circuits that are necessary because of the nature of visual circuit building. On the other hand, the representation contains little information about the behavior of the circuit. For example, for a given input pin, it is non-trivial to discover source of the signal.



a : Input	1: $a \rightarrow b$
b : Joint	2: $b \rightarrow c$
c : Joint	3: $c \rightarrow d$
d : Output	

Figure 29: In-editor representation of trivial circuit

To demonstrate the inefficiency inherent in the representation, look at Figure 29. Say we need to figure out the state of the ‘output’ component’s input pin. To do this, we start at **d** and look for wires connected to **d**. We find **3** to be connected to **d**, and see that it’s also connected to **c**. Then for **c** we again look for wires, etc. etc. It is obvious that a different representation is advantages. The same circuit’s representation in IR can be seen in Figure 30. How it works will be explained shortly.

```
component <main> 1 1 {
  state [ #0 ]
  %0 = Input 0
  SetState #0, %0
  Output 0, %0
}
```

Figure 30: IR representation of trivial circuit

The IR is a *definitive* representation defined as a *Linear IR*.

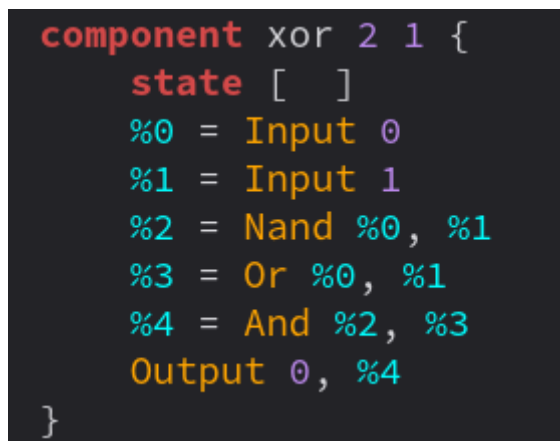
*Linear Irs represent the program as an ordered series of operations. [...] The linear IRs used [...] often resemble the assembly code for an abstract machine.*³⁵

³⁵ Engineering a Compiler, p. 175

The IR, or rather it's textual representation, can be described with the following BNF grammar:

```
circuit ::=
component ::= "component" ident int int "{" state-decl stmt-list "}"
state-decl ::= "state" "[" state-list "]"
state-list ::= state | state-list "," state

stmt-list ::= stmt | stmt-list stmt
stmt ::= reg "=" rvalue-mnemonic op-list
      | lvalue-mnemonic op-list
op-list ::= op | op-list "," op
op ::= reg | state | int | call-op
call-op ::= ident "(" op-list ")"
rvalue-mnemonic ::= "Null" | "Input" | "GetState" | "Call" | "Elem"
                | "Not" | "And" | "Or" | "Nand" | "Nor"
lvalue-mnemonic ::= "Output" | "SetState"
reg ::= "%" int
state ::= "#" int
int ::= /[0-9]+/
ident ::= /[a-zA-Z][a-zA-Z0-9]*/
```



```
component xor 2 1 {
  state [ ]
  %0 = Input 0
  %1 = Input 1
  %2 = Nand %0, %1
  %3 = Or %0, %1
  %4 = And %2, %3
  Output 0, %4
}
```

Figure 31: Example of circuit of XOR component

The IR for a circuit is a list of components represented by instances of the distinct **Component** class.³⁶ A **Component** instance contains a list of **Stmt** objects, a list of **State** objects, the amount of

³⁶ Defined in `app/src/editor/ir.ts` (Note that these types are distinct types from those defined in `./board.ts`)

inputs and outputs, and a string label. **Stmt** contains it's variant (**StmtKind**) including the operands for the statement.

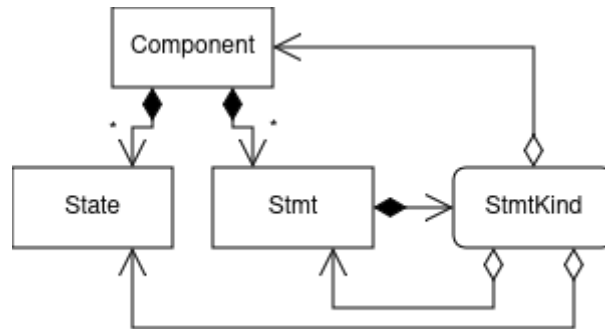


Figure 32: Class diagram of IR

The **StmtKind** type is defined as a union type of each variant:

```

export type StmtKind =
| { tag: "Null" }
| { tag: "Input"; i: number }
| { tag: "Output"; i: number; src: Stmt }
| { tag: "GetState"; state: State }
| { tag: "SetState"; state: State; src: Stmt }
| { tag: "Not"; src: Stmt }
| { tag: "And" | "Or" | "Nand" | "Nor"; lhs: Stmt; rhs: Stmt }
| { tag: "Call"; comp: Component; inputs: Stmt[] }
| { tag: "Elem"; src: Stmt; i: number };
  
```

The *Null* kind represents an unconnected pin. Example:

```
%0 = Null
```

The *Not*, *And*, *Or*, *Nand* and *Nor* components represent the primitive boolean operations, they take one or two operands and represent the resulting value. Example:

```
%2 = And %0, %1
```

The *Input* and *Output* pins are associated with the 'input' and 'output' components, or input and output pins, of a circuit. The 'i' is an index into the given input-array or resulting output-array.

Example:

```

%0 = Input 0
SetState #0, %0
%1 = GetState #0
Output 0, %1
  
```

Call and *Elem* are used to represent custom components. The call contains the IR representation of the component and an array of input-values. The *Call* statement represents the output-array. The *Elem* statement is to retrieve specific output values. Example:

```
%0 = Input 0
%1 = Input 1
%2 = Call my_component (%0, %1)
%3 = Elem %2, 0
Output 0, %3
%4 = Elem %2, 1
Output 1, %4
```

The state system with **State**, *GetState* and *SetState* has two purposes. The first purpose is to represent actual state in a circuit. State in a circuit is formed by cycling connections. The specific activation of a cyclic circuit needs to be preserved across simulations. Each state, represented textually by #0, #1, ... is saved across simulation runs.

The second purpose of the state system is to show activation of wires. The way the **Board** knows which wires have been activated are by associating each wire with a state. After a simulation run, the **Board** can query for the states associated with the wires. See for example Figure 30 for a state that is unnecessary for the circuit behavior itself, but used to activate a wire.

The IR is in *static single-assignment* (SSA) form.

*In SSA form, each name corresponds to one definition point in the code. The term static single assignment refers to this fact.*³⁷

This has some advantages. Mainly the advantage is that operand refers to a specific definition, most crucially, each operand of a statement refers to a specific other statement in a strictly linear fashion.

*As a corollary, each use of a name in an operation encodes information about where the value originated; the textual name refers to a specific definition point.*³⁸

This is highly desirable, because that means that tracing the signal for activation of a certain ‘output’ component is trivially done by just looking at the operand’s definition.

The textual representation of the IR uses named registers, such as %0, %1, ... to refer to other statements. In the actual IR, these references are direct references to the statements.

³⁷ *Engineering a Compiler*, p. 193

³⁸ *Engineering a Compiler*, p. 193

Circuit lowering

Circuit lowering is implemented in the **toIr()** function in **Board**. The algorithm uses the *Visitor pattern*³⁹ to traverse the in-editor circuit representation. Using a visitor pattern here serves to separate the traversal logic from the lowering logic of the algorithm, simplifying both.

A visitor gathers related operations and separate unrelated ones. Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor.

*Unrelated sets of behavior are partitioned in their own visitor subclasses. That simplifies both the classes defining the elements and the algorithms defined in the visitors. Any algorithm-specific data structures can be hidden in the visitor.*⁴⁰

I experience that it was the case, that this simplified the code significantly, compared to my original approach of building the lowering algorithm into the data structures themselves.

The core concept of the lowering algorithm is that each wire initially will be assigned its own state-object. Each joint and statement is lowered, with the input and output lowered to a *GetState* or *SetState* on the state-objects for the connected wires. All joints and statements are lowered directionally starting from each 'input' component. From each 'input' component, the flow traverses through each connected wire, joint and component, in the same way as signals would flow. This directional flow is not strictly necessary. It is implemented like this, because it was necessary at an earlier point. Each item is added to a set, to ensure that they are only lowered once. This is relevant for cyclic connections.

Lowering uses the **ComponentBuilder** class⁴¹ to instantiate statements, state-objects and the final component. The **ComponentBuilder** stores each instance of statements and state-objects, so that the resulting components have a list of each. The **ComponentBuilder**'s factory methods also return a reference to the instances. This is so that the lowering algorithm can connect items by referring to them.

39 *Design Patterns*, p. 331

40 *Design Patterns*, pp. 335-6

41 Defined in *app/src/editor/ir.ts*

Optimizing IR

The next step is optimizing the IR generated from lowering. This is needed primarily because the IR generated is inefficient. The lowering algorithm constructs the IR quite literally to the in-editor representation.

The primary deficiency of the algorithm is that it generates unnecessary state-objects. For each wire a state-object is generated. And all connections are generated through these state-objects. To see why this is problematic, see the circuit in Figure 33 and the unoptimized and optimized IR in Figure 34.

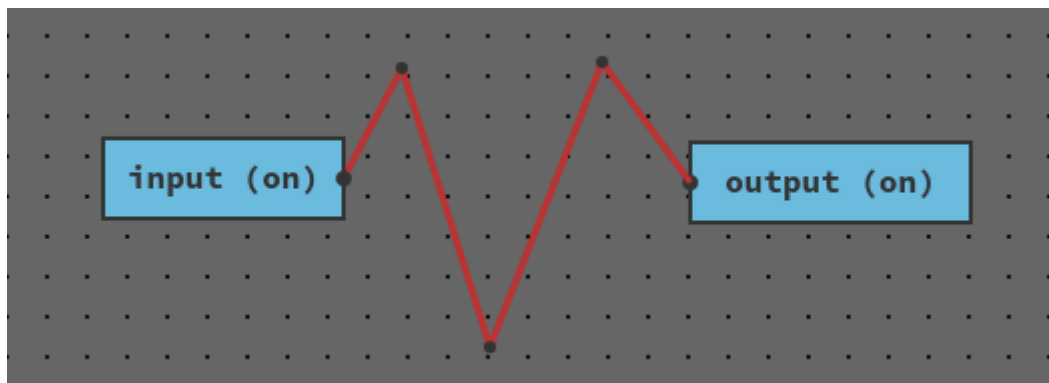


Figure 33: Simple input-output circuit with 3 joints

The unoptimized IR generates a state-object, and subsequent *SetState* and *GetState* statements for each of the wires. For a trivial circuit like this, there is no reason for this.

If a *GetState* immediately follows a *SetState* on the same state-object, the operation is superfluous. The two statements can therefore be removed, and the source for the *GetState* statement can instead just be used directly. Using this and other heuristics, where behavior can be guaranteed to be preserved, the optimizer does various optimizations to improve the IR.

Using linear IR in SSA form is very beneficial here, because it's relatively uncomplicated to remove these redundant operations; they are apparent just by looking at the representation, and the solution is also apparent.

```
Before optimizing

component <main> 1 1 {
  state [ #0, #1, #2, #3 ]
  %0 = GetState #0
  Output 0, %0
  %1 = Input 0
  SetState #3, %1
  %2 = GetState #3
  SetState #2, %2
  %3 = GetState #2
  SetState #1, %3
  %4 = GetState #1
  SetState #0, %4
}

After optimizing

component <main> 1 1 {
  state [ #0 ]
  %0 = Input 0
  SetState #0, %0
  Output 0, %0
}
```

Figure 34: Input-output circuit IR before and after optimizing

The IR optimization is implemented in the **ComponentOptimizer** class.⁴² The optimizer is implemented as a loop over a series of passes. Each pass looks for a certain conditions, and if the condition hold true, mutates the IR. After a certain amount of iterations, the optimization is deemed good enough.

Separate passes may depend on each other, e.g. one pass may incur a change that enables another pass to incur a change and so on. The optimizer therefore has a policy of deciding when to exit the optimization loop. This is implemented with a ‘score’ that is calculated as an aggregate of a component. The calculation of this score is arbitrary and based on heuristics. In the optimizer, the calculation is $N_{components} \cdot 100 + N_{states}$. Through testing, this has shown itself to be adequate.

Because **Board** uses state-objects for wires, each wire’s state-object has to be preserved, or updated to reflect optimizations. One optimization called **collapseStates** merges two states, if they are

⁴² Defined in `app/src/editor/ir.ts`

identical in regards to the behavior of the circuit. When such optimizations occur, **Board** needs to know, so that it can update the wires using the discarded state-object with the preserved one. This is done by maintaining a list replaced states. After optimization, **Board** will iterate through this list and replace wires' state-objects accordingly.

For the *main* circuits, i.e. the one being simulated, the wire states need to be preserved. But for circuits that are used as components in the main circuit, those don't have any wires they need to activate and deactivate. The states-objects that are unused internally in these circuits can therefore be removed. To support this difference in optimizations, **ComponentOptimizer** has 2 entry points: **optimizeMain()** and **optimizeComponent()**.

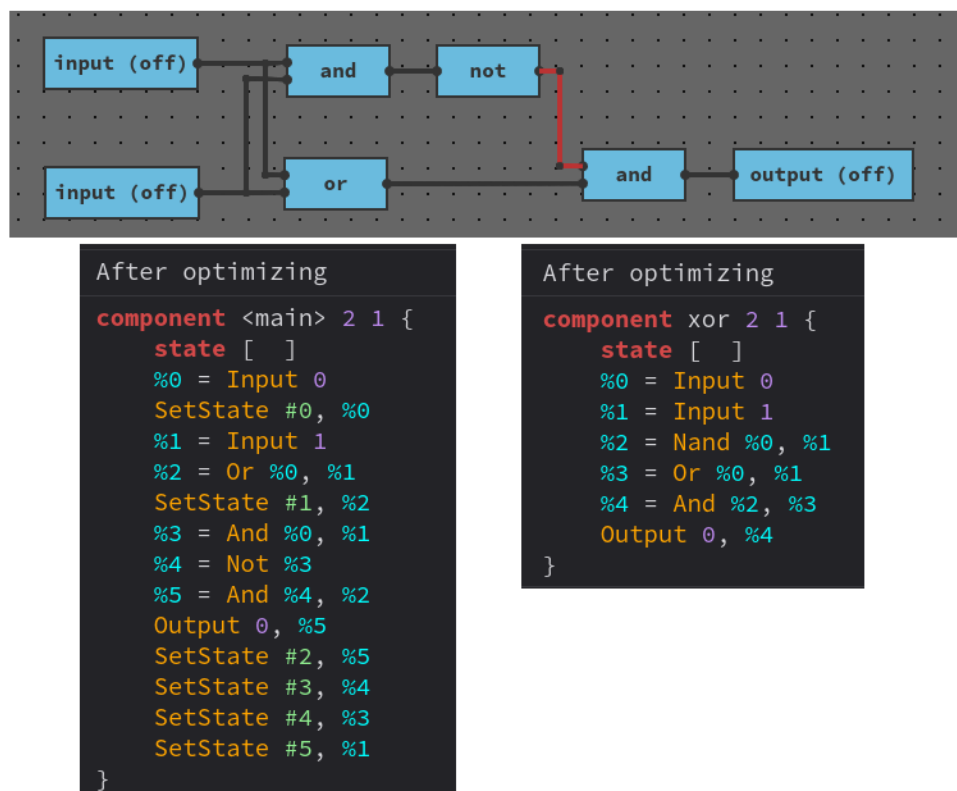


Figure 35: Difference between `optimizeMain` and `optimizeComponent`

The optimization passes are:

- **eliminateRedundantState:** If we get a value from a state, that has been set earlier, then just use the value that set the state originally.

- **hoistInputs:** Move *Inputs* so they're the first statements in the component. This enables further optimizations.
- **moveSetStateToSource:** Move *SetStates* to right below their source. This also enables further optimizations.
- **collapseStates:** If two states behave the same, remove one of them, and only use the other.
- **eliminateUnusedStates:** If a state-object is unused, discard it.
- **eliminateRedundantSetState:** If a state is set twice without being used in between, remove the first *SetState*.
- **eliminateIndependentSetState:** (Only for component-circuits) If a state-object is never read, remove every *SetState* with that state-object.
- **rewriteToNandOrNor:** AND and OR gates immediately followed by NOT gates can be rewritten to NAND and NOR gates respectively.
- **eliminateUnusedStmts:** Remove any statement that is neither used nor produces side effects.

Simulation

The simulation is encapsulated in the **Sim** class.⁴³ The simulator takes as input an IR component, an input-array, an output-array and a state-map. The output-array is an out-parameter that will contain the result after simulation. The state-map is given as a parameter, because the map's lifetime is bounded to the **Board** instance, not the **Sim** instance.

The simulator uses *boolean* values to represent signal-state. The input and output arrays are arrays of booleans and the state-map is a mapping from state-objects to booleans.

The simulator is implemented as a loop that iterates through the statements of the IR component. For each statement, it will run the behavior corresponding to the statement's kind. The simulator transfers values using the statements as though they are registers with a continuous associative array pattern using an array and an index map.

For evaluating custom components, an inner **Sim** instance is instantiated. The values of the input-pins are converted to an input-array to the new instance, and the state-map is passed in as well. The output-array will then become the *Call* statement's result.

One of the non-functional requirements is:

⁴³ Defined in *app/src/editor/sim.ts*

Simulating interactively without unreasonable delay. The system should be able to simulate any circuit with less than 100 components in less than 500 milliseconds on modern hardware.

By implementing the simulator as a sequential loop over a linear IR, the simulator is significantly more efficient computationally than simulating the in-editor representation directly. For any circuit that I have built during development, simulation has seemed instantaneous. Thus this requirement is fulfilled.

As an alternative to the current approach, I contemplated making a native application in C++ that would serve as the runtime for circuits. This way, I could have implemented a simulator that was significantly faster. Due both to time constraints and the satisfactory result of implementing the simulator as it is now, I decided against it.

UI, IO, and rendering

The application is implemented as a Vite/React web application. The UI architecture consists of having a React app with some React components. Each of the toolbar, tab-bar and editor-canvas are represented as distinct React components. The **App** component⁴⁴ is the conceptual root of the application. Here, an instance of **Editor** is instantiated. The **App** component uses the **Toolbar**, **Tabbar**, and **Canvas** components.⁴⁵ A reference both to the editor and to the canvas is distributed across the React components.

The **Canvas** component uses an HTML canvas for the purpose of rendering the editor onto it. Input handlers are attaches that each send distinct events to the editor event bus. Also in this component is a React effect that subscribes to the event bus, that subscribes on the *RenderRequest* event. When this event is received, the **Canvas** component will ask the instance of **Editor** to render itself on the HTML canvas.

Each object that is visible on the screen is responsible for rendering itself. They therefore have a **render()** method. The rendering calls are structured hieratically so that it's easy to specify the order of rendering. The canvas drawing itself is encapsulated in the **Renderer** class.⁴⁶ An instance of this

⁴⁴ Defined in *app/src/App.tsx*

⁴⁵ Defined in *app/src/{Toolbar,Tabbar,Canvas}.tsx*

⁴⁶ Defined in *app/src/editor/Renderer.ts*

class will be passed around to each **render()** method. Generally, objects pass positions and data to the renderer, and then it's the renderer's responsibility to determine how these positions and data should look visually. For example, **Board** calls **Renderer's drawComponentBodySelectedInternal()** with a position and a component-kind that tells the size and label. The rest is up to the renderer to figure out.

Sequence diagrams

Not that the interactions in the system are more complex than depicted on the sequence diagrams. The diagrams are aid in understanding, not to specify interactions in detail. Due to the event-driven architecture, the sequences of events in the program are largely sporadic. An attempt to diagram every interaction is a hopeless endeavor.

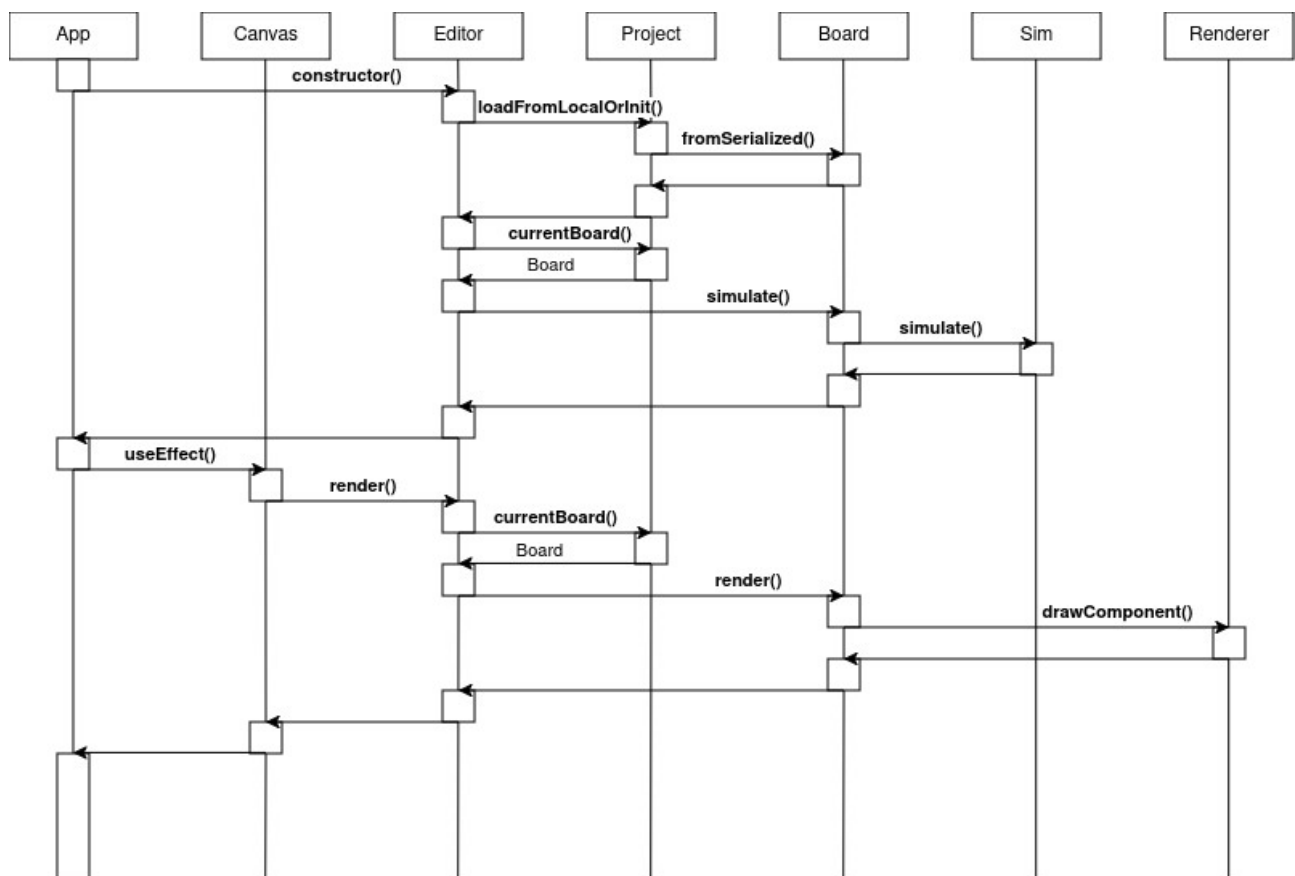


Figure 36: Sequence diagram when the application initializes

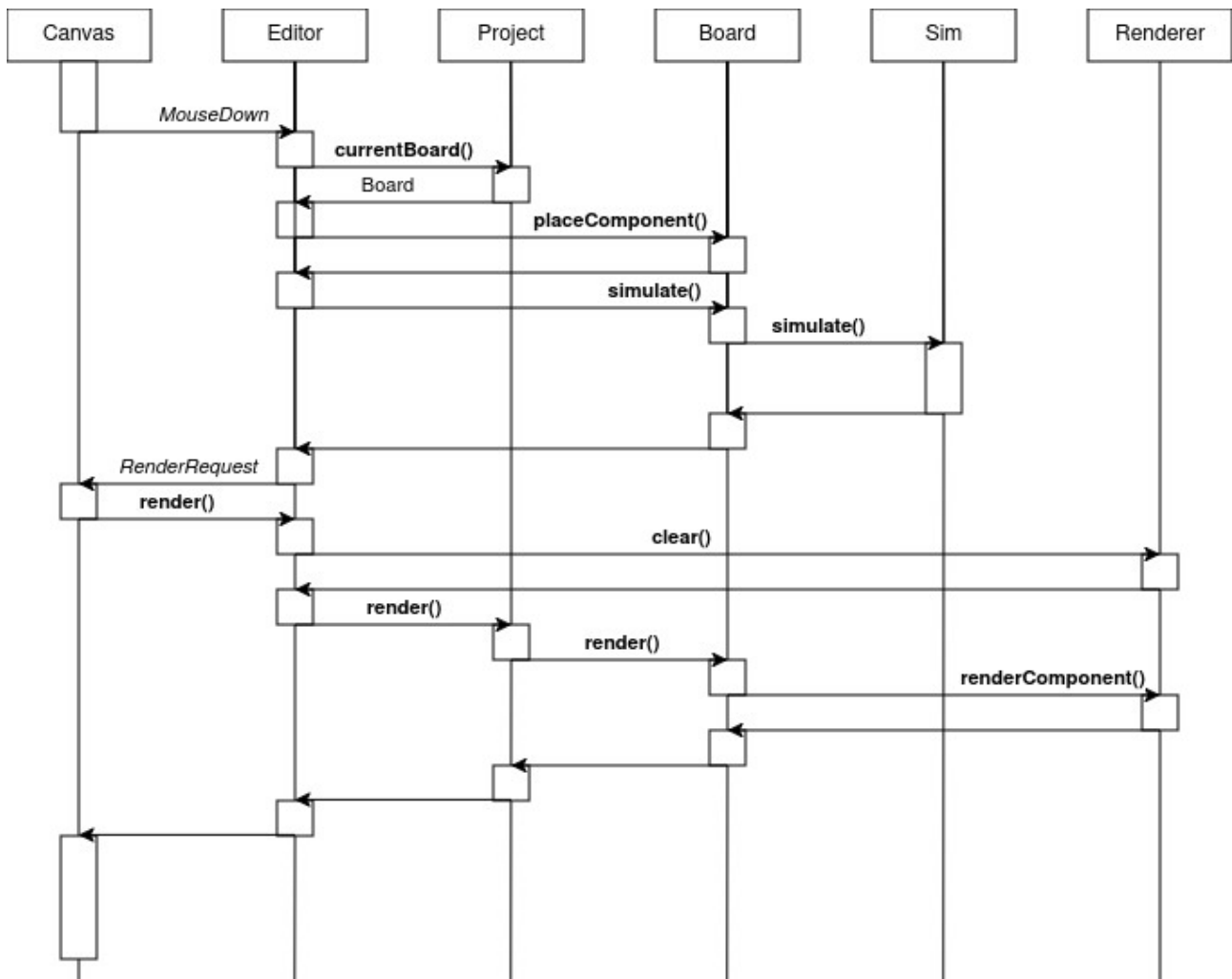


Figure 37: Sequence diagram of a mouse click that places a component and subsequently triggers a re-render

Class diagram

The following is an almost complete UML class diagram of the application. The arrows state the relationship of the classes with *owns*, *references*, and *implements* relations. This class diagram can also be found in the docs/ folder.

I have decided not to include methods or fields for the simple reason that there would be too much information. I have attempted in earlier sections to describe the state and interactions of the system, and I deem that this as sufficient as I can do it.

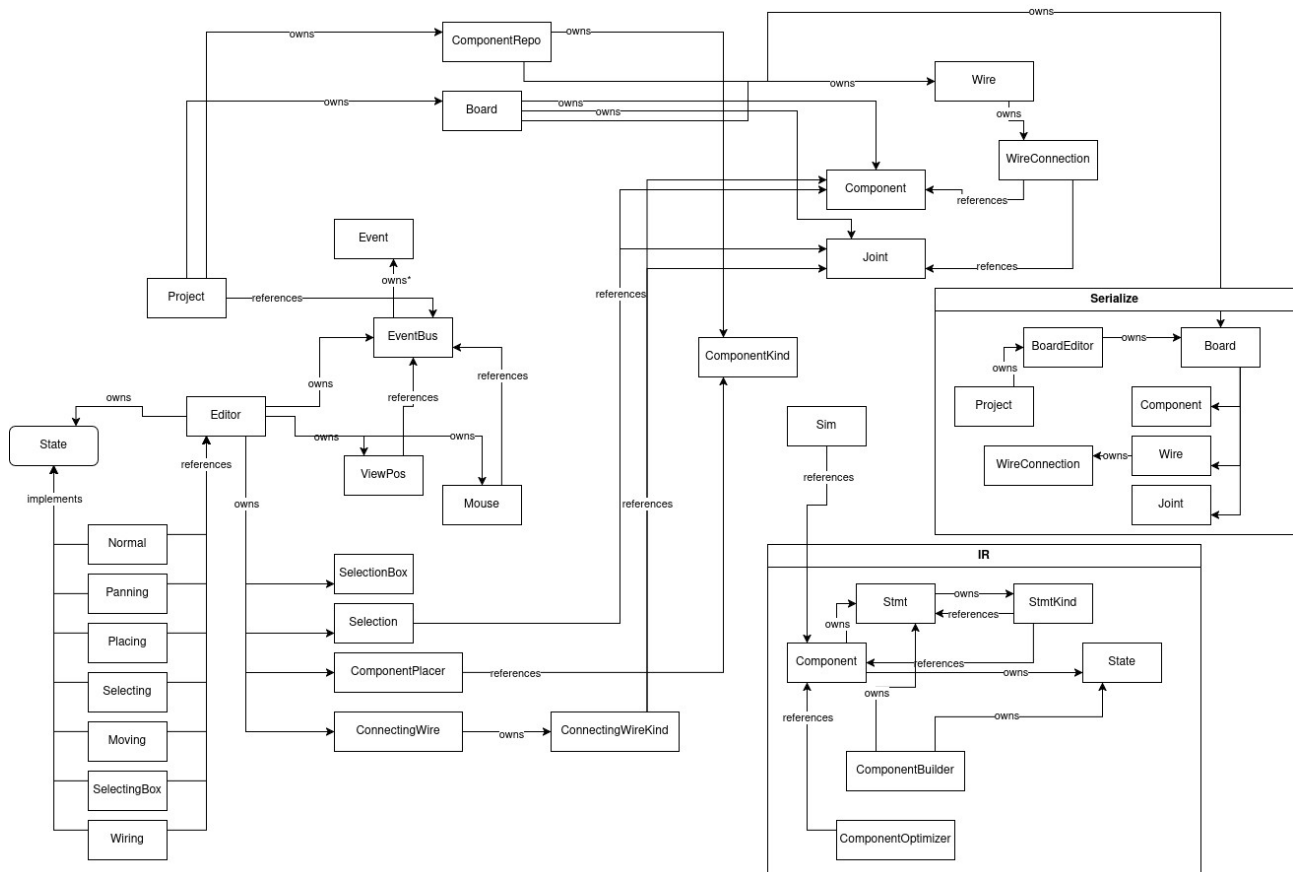


Figure 38: Class diagram

Literature

John English: *Introduction to Operating Systems*, PALGRAVE MCMILLAN, 2005, ISBN 0-333-99012-9

Andrew S. Tanenbaum, et al.: *Distributed Systems – Principles and Paradigms*, PEARSON, 2014, ISBN 1-292-02552-2

Joe Armstrong: *Programming Erlang – Software for a Concurrent World*, The Pragmatic Programmer, 2007, ISBN 1-9343560-0-X

Erich Gamma, et al: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN 0-201-53361-2

Keith D. Cooper, Linda Torczon: *Engineering a Compiler*, Third edition, Morgan Kaufmann, 2023, ISBN 978-0-12-815412-0