DLS

The digital logic simulator game

Sandbox mode manual

v0.16.3

2019-01-27

Table of Contents

Introduction

Welcome reader! Since you are reading this I assume you already know what DLS is about and you are searching for information on some part of the UI. So I'll keep this as short as possible.

DLS is a "**time-driven event-based multi-delay 3-value digital logic simulator**" (if there's such classification). Let's try to break it up in pieces to better understand how it works:

- **Time-driven** means that the circuit time is advanced forward based on a user-specified target speed, which is measured in circuit nanoseconds per real second (ns/s). The simulation **always advances** to a new state if there are pending signals in the circuit's queue. This means that even **unstable** or **asynchronous** circuits can be correctly simulated.
- **Event-based** means that **a gate is simulated only if one of its inputs changes value**. Otherwise the previous output is considered valid and used as input to all connected gates/components.
- **Multi-delay** means that each build-in gate/component type has its own **propagation delay** which is always an integer greater than or equal to 1.
- **3-value** means that in addition to the 0 and 1 logic levels, there's an extra logic value (U for Undefined) which is used as either Z (high impedance) or X (undefined) signals, depending on its origin.

The target simulation speed can get as low as 1ns/s, which can be used to visually debug signals propagating through a circuit, and as high as 1s/s, in case you want to let it run as fast as possible. Note that the target simulation speed might not be achieved so the real simulation speed will differ. It depends on the complexity of your circuit, but it shouldn't affect the final results.

What might affect your results is the selected clock speed, in case your circuit is a clocked (sequential) circuit. You have to take care to use a slow enough clock to allow your signals to arrive and stored at the destination registers before the next clock tick is triggered. Clocks can go as slow as 1Hz and as fast as 500MHz. Note that the clock speed is measured in simulation time and not real time.

E.g. a 500MHz clock simulated at a target speed of 1ns/s will switch levels every second. The same is true for a 1MHz clock with a target speed of 500ns/s. On the other hand, a 10MHz clock on a 100us/s simulation will switch levels 2000 times during 1 real second (assuming your circuit is simple enough to achieve the target simulation speed).

That's all you need to know about how the simulator works. Now go and make something great :)

1. User Interface

The image above shows DLS' main user interface (UI), with both the right and bottom panels opened.

1. Main toolbar
2. Right panel: Simulation, Testbench
3. Bottom panel: Console, Logic Analyzer
4. Gate/component toolbar

In the following sections, the various parts of the UI will be described.

1.1 Main toolbar

The main toolbar includes the following buttons, from left to right:

- **Start/Stop simulation**: When enabled (pause symbol is visible) every time the circuit changes (e.g. input values change, wires added or removed, etc.), outputs are recalculated. When disabled (play symbol is visible) no simulation is performed. For example, stopping the simulation is an easy way to change multiple input values at once.
- **Save schematic**: Save the current schematic on disk (*).

- **Load schematic**: Load a previously saved schematic from disk (*).
- **Componentize circuit**: Create a new component from the current schematic. For details on how components work see chapter 3.
- **Clear schematic**: Clears the current schematic.
- **Menu**: Show the pause menu from which you can return to the main menu or exit DLS.

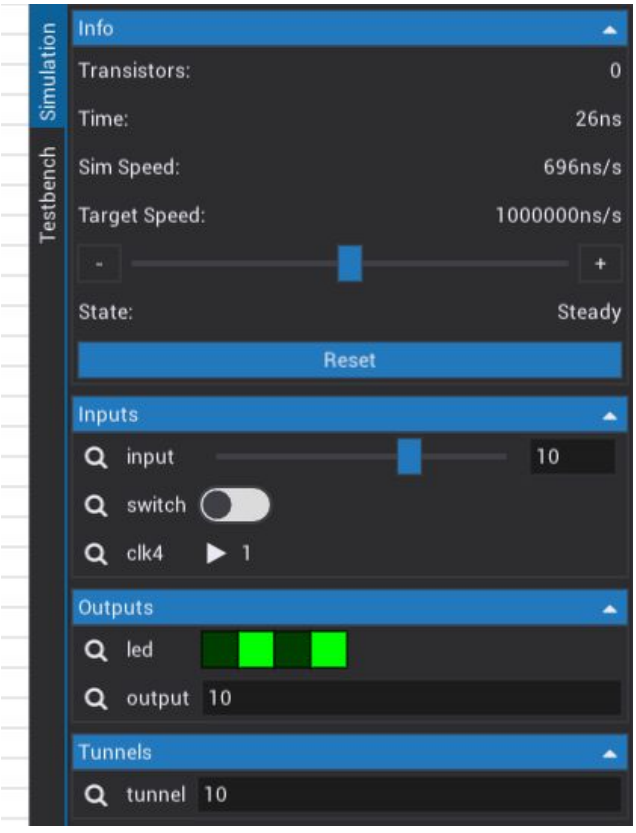
(*) Depending on your platform, schematics are saved at the following locations:

- Windows: %USERPROFILE%\Documents\DLS\schematics (your Documents folder)
- Linux: **Locations changed in v0.16.2**.
  - If `XDG_DATA_HOME` environment variable is defined, the path is `$XDG_DATA_HOME/DLS/schematics`.
  - Otherwise, the path is `$HOME/.local/share/DLS/schematics`.
  - If the home folder cannot be determined, schematics are saved under the `schematics` subfolder on your installation directory.
- OS X: ~/Documents/DLS/schematics (your Documents folder).

1.2 Right panel

The panel on the right includes 2 tabs: Simulation and Testbench. You can open each tab by clicking on it. You can close the panel by clicking again on the open tab.

1.2.1 Simulation



The image on the left shows the simulation tab. It includes 4 foldable sections: Info, Inputs, Outputs and Tunnels.

Info displays information about the current circuit, such as the number of equivalent CMOS transistors, the current circuit simulation time, the current simulation speed (calculated from the last simulation loop), the target simulation speed and the circuit's state.

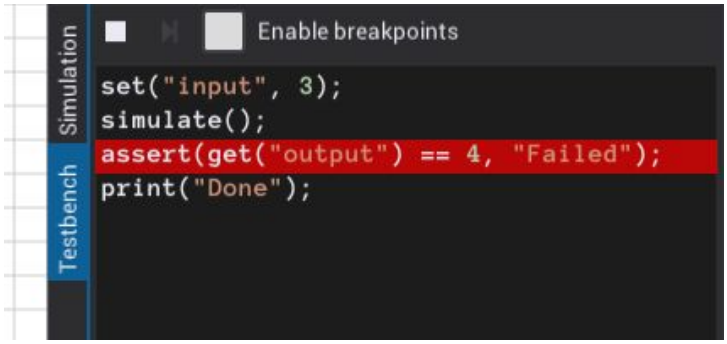
State can be either Steady or Transient. Steady means that the circuit has finished processing its event queue and all outputs will remain unchanged until an input changes value. Transient means that there are more events queued to be processed at a later

timestep and outputs will probably change until the circuit returns to steady state.

Finally, the Reset button resets the circuit simulation time and clears the event queue.

Inputs, **Outputs** and **Tunnels** show the respective circuit components, which you can control from there. You can also click the little magnifying glass icon next to them to find them in the circuit.

1.2.2 Testbench



The Testbench tab has the testbench editor and its corresponding controls.

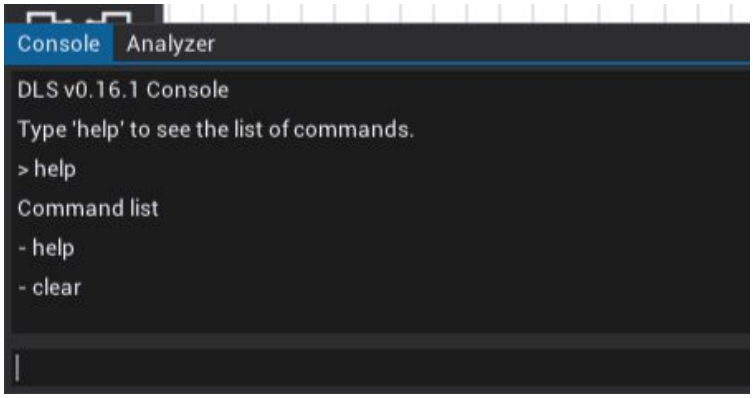
Testbenches are Lua scripts from which you can control inputs and observe outputs.

On the top there are 2 buttons and a checkbox. The first button starts and stops the testbench. The second button resumes testbench execution until the next breakpoint, if breakpoints are enabled. For more details on testbenches see chapter 5.

1.3 Bottom Panel

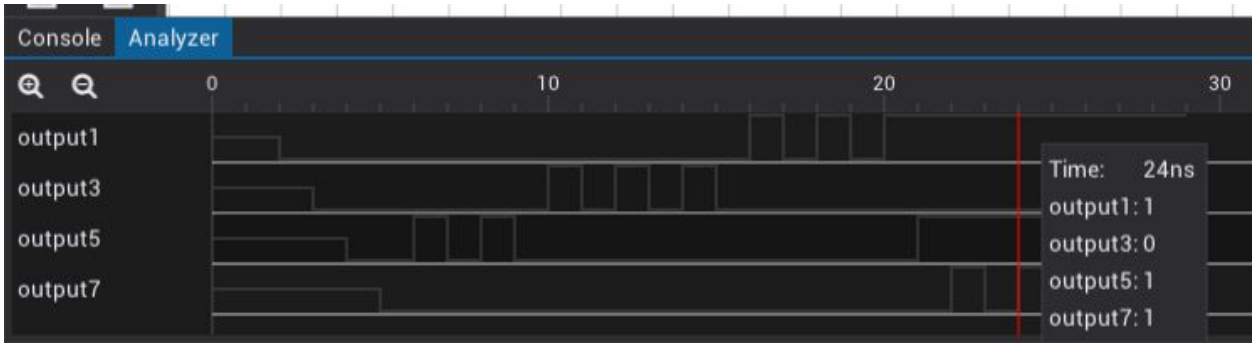
The panel at the bottom of the screen includes 2 tabs: the Console and the Logic Analyzer.

1.3.1 Console



The image on the left shows the console. You can use the console to print debug messages from scripted components or testbenches.

1.3.2 Logic Analyzer



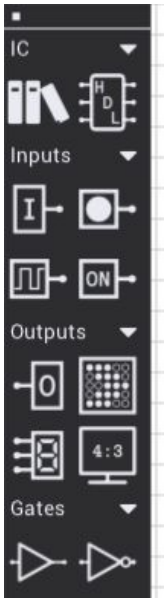
The logic analyzer collects and displays the values of all **debug numeric outputs** through time. In order to monitor the value of a wire/output pin you have to connect a debug output port to it. The analyzer can be used to find delays in your circuits and for debugging their behavior (finding hazards and glitches).

Clicking the Reset button from the Simulation tab resets the history of all the debug outputs, as well as the logic analyzer timeline.

The two buttons at the top left of the timeline are used for zooming in and out. You can select around which timestep you want to zoom by clicking anywhere in the timeline (a blue bar will appear). Alternatively, you can hold down the Ctrl button on your keyboard and zoom using the mouse wheel, around the mouse cursor.

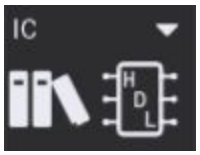
Finally, you can use the keyboard to navigate the timeline. Select any row in the timeline and use the left and right arrow keys to jump to the previous or next time the value of this output changed. You can also change the selected row by using the up and down keys.

1.4 Gate/component toolbar



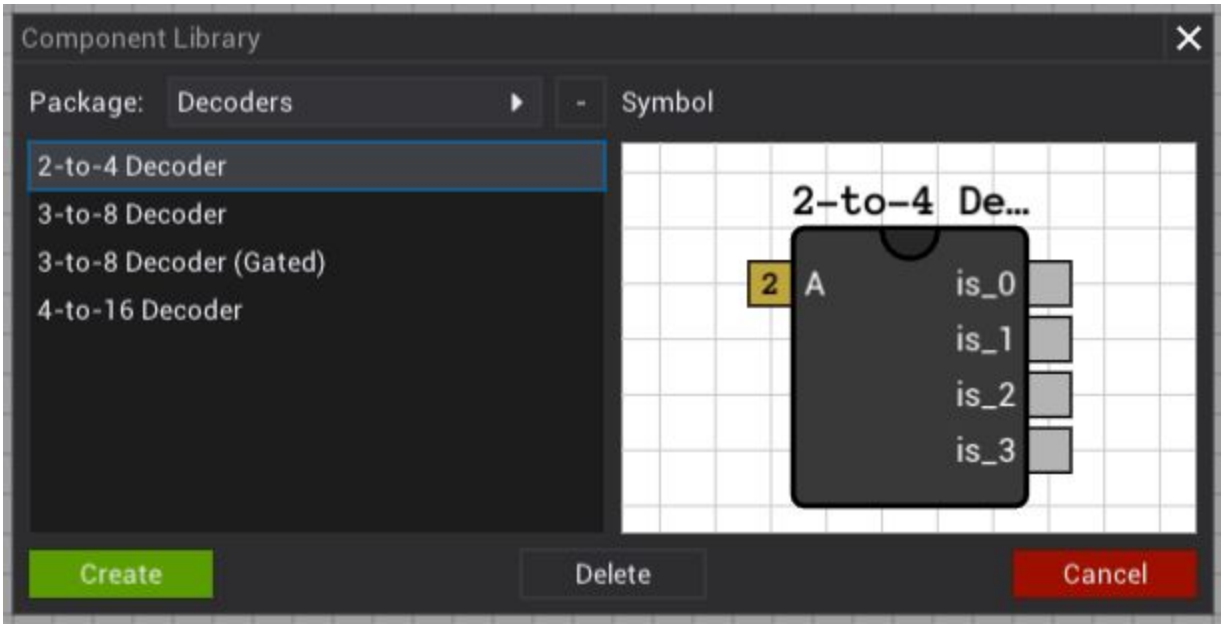
The image on the left shows a part of the gate/component toolbar (from now on just gate toolbar). From v0.15 and up, the gate toolbar has changed compared to previous versions and now all available options are grouped based on their functionality. Clicking on the group name will toggle the visibility of the group's options.

1.4.1 IC



The IC group includes the following options (left to right):

1.4.1.1 Component library



The image above shows your component library. Components are separated into packages. When starting DLS for the first time, a default empty package is already created for you, which you cannot delete.

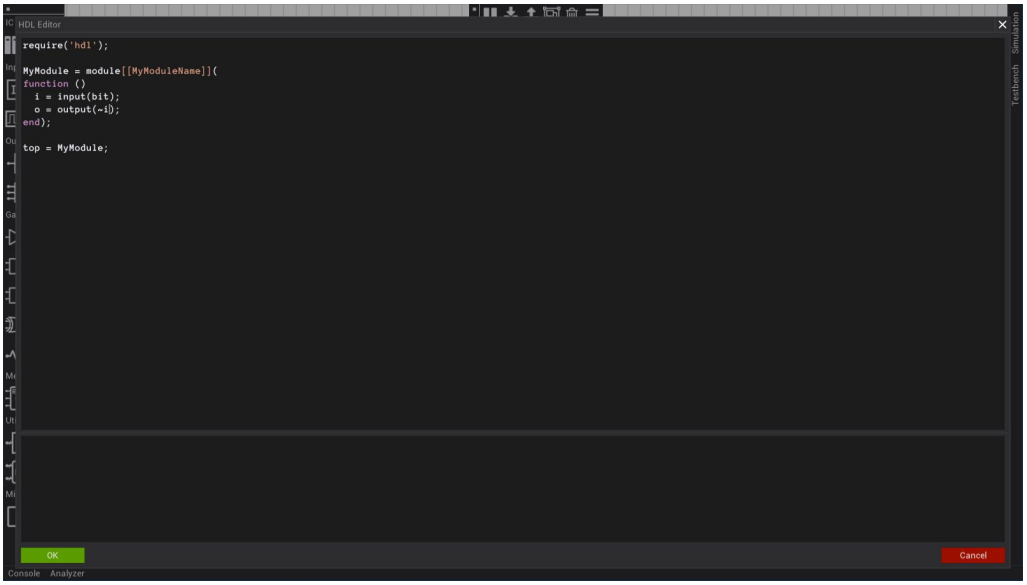
The list on the left side shows the components from the selected package. On the right side there's a preview of the selected component. You can zoom and pan the preview using the mouse, just like the schematic view.

Clicking Create will create a new instance of the selected component, which you can place on the grid. The Delete button removes the selected component from the library and Cancel closes the dialog.

Right clicking on a component already placed in the circuit brings up extra options. One of them is Inspect which lets you inspect the component's circuit.

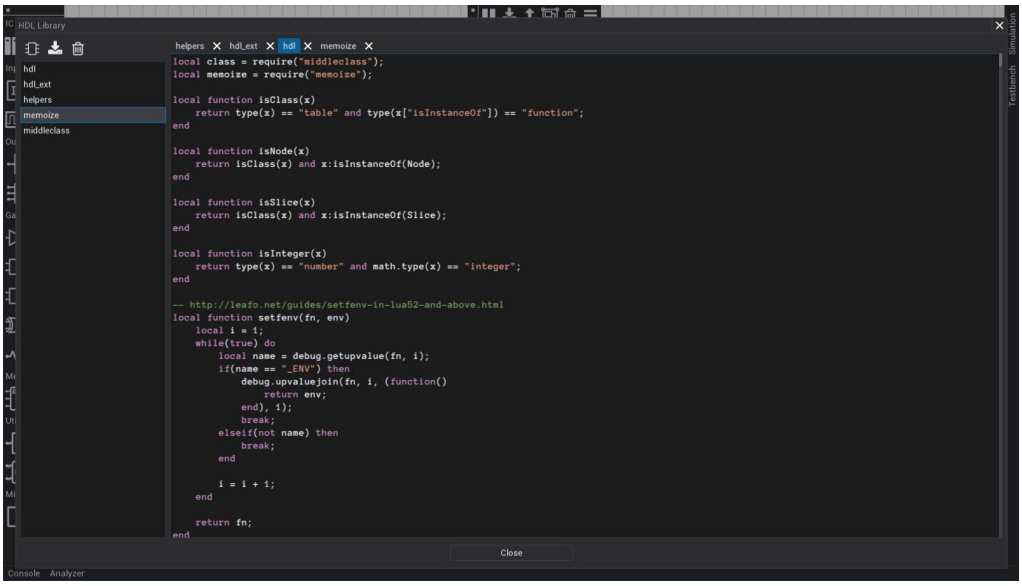
Note: Currently there's no way to edit a component. If you want to change the internals of a component you have to re-componentize the circuit from the Main toolbar, once you've made your changes, and manually replace the component in the parent circuit.

1.4.1.2 Hardware Description Language (HDL) components



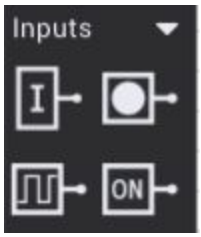
From v0.16 and up, DLS includes a custom HDL based on Lua. For more info on the HDL syntax check chapter 6.

Left clicking on the HDL icon brings up the HDL component editor (shown above). The editor is split into two parts. The top part is where you write your code. The bottom part is used to display potential errors. If there are no errors, clicking OK will create a new instance of your component. Clicking Cancel closes the editor. Note that all changes will be lost if you click Cancel.



Right clicking on the HDL icon brings up the HDL library (shown above). You can store reusable HDL code in separate files here and include them in your HDL components. The HDL library files are stored in the user data folder under the 'hdl' subfolder.

1.4.2 Inputs



The Inputs group includes all available input sources.

1.4.2.1 Numeric Input

The image on the left shows the configuration dialog for a new numeric input. You can open this dialog by right clicking on the toolbar button. The first parameter is the width of the input port in bits.

If the **Is Bus** checkbox is checked, the new input port will have 1 wide pin. Otherwise, it'll have multiple 1-bit pins.

Const means that the input port is a constant value and it won't be part of the generated component. Constant inputs can be used to set signals to predefined values.

You can interact with the numeric input port by either left clicking on its value (increases the value by 1), or by using the mouse wheel while the cursor is over the value (scrolling up or down increases or decreases the value, respectively).

Finally, numeric inputs have multiple display formats. Depending on the width of the input these are: unsigned (u), signed (s), hex (h) and ASCII (a) ('a' is available only for 8-bit inputs). You can cycle through the various formats by clicking on the small letter at the top left corner of the input port.

1.4.2.2 Push button

Push buttons are 1-bit input ports. Their only configurable parameter is whether they will be Const or not. When placed in a schematic, you can interact with the push button using your mouse. As long as the button is pressed, its value will be 1. Otherwise, it'll automatically revert to 0.

1.4.2.3 Clocks

Clocks are 1-bit input ports which oscillate between 0 and 1 when activated. The image on the left shows the clock configuration dialog.
The only available parameter is the duration of the half clock cycle (T/2), measured in nanoseconds. The minimum clock frequency is 1Hz (tick every 500ms) and the maximum clock frequency is 500MHz (tick every 1ns).

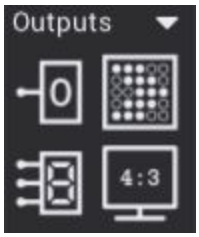
You can change the default clock frequency by using the slider. Dragging the handle to the left decreases the value. Dragging it to the right increases it. The dragging distance affects the rate of change of the parameter.

Clocks can be either stopped or running. Finally, clocks cannot be marked as Const.

1.4.2.4 Switches

Switches are multi-pin input ports. A switch has multiple 1-bit pins which you can control using the individual switch controls inside the port. Const has the same meaning as with the other input port types.

1.4.3 Outputs

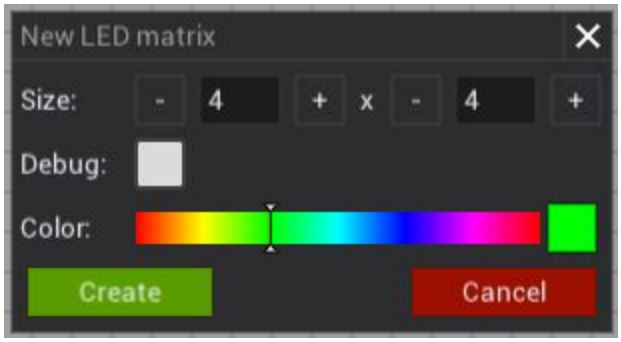


The Outputs group includes all available output ports:

1.4.3.1 Numeric Output

Similar to the numeric input port is the numeric output port. The only difference is that instead of being **Const**, output ports are marked as **Debug**. When **Debug** is checked the output port won't be part of the generated component. Debug numeric output ports can be used to observe intermediate signals using the Logic Analyzer.

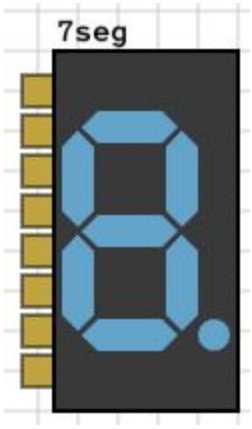
1.4.3.2 LED Matrix



A NxM LED matrix contains M rows by N columns of LEDs. There's one N-bit pin for each one of the M rows. The maximum size of an LED matrix is 16x16.

The color picker can be used to select the On color of the LEDs. The Off color has the same hue with a lower brightness.

1.4.3.3 7-segment display



7-segment displays are another type of output port. They have 8 1-bit pins (1 for each segment plus a pin for the dot) as shown on the left. Whenever a pin is HIGH, the corresponding segment is turned on. Otherwise it remains off.

As an output port it can be configured by right clicking on the toolbar button. The only configuration option is whether the port will be a debug port or not. Note that even if the 7-segment display is marked as Debug, its value will not appear in the Logic Analyzer.

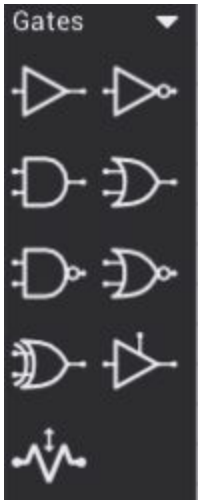
1.4.3.4 Pixel display

The left image above shows the configuration dialog of a Pixel Display and the right image shows an example of a 320x200 display.

Displays are sequential components (i.e. they have a clock input) with internal memory. The *ax* inputs combined make up the address of the first pixel you want to write to. *D* is the data of the 8 pixels you want to write and *clk* is the clock. Whenever *clk* goes from LOW to HIGH, the data found at the *D* input are written to the corresponding pixels, starting at address *A* (LSB of the address is the LSB of *A0*).

Currently, displays support only 1bpp. You can clear the display to black or a random color for each pixel via its context menu.

1.4.4 Gates

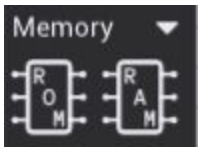


The Gates group includes all available logic gates. Left clicking on any one of those will place a new instance under the mouse cursor. Right clicking opens the configuration dialog. From there you can configure the number of inputs as well as the number of bits for the gate. AND, OR, NAND, NOR have a variable number of inputs (from 2 to 16). Buffer, NOT, XOR, Tri-state buffers and Pull Resistors have a fixed number of inputs.

For a brief explanation on the purpose of each gate, place the mouse cursor on the corresponding toolbar button and wait for the tooltip.

Buffers are the only gates with configurable propagation delay. All other gates have standard propagation delays, which depend on their number of inputs. For a complete list see Appendix A.

1.4.5 Memory



The Memory group includes all available memory components. Currently those are Static RAMs and ROMs.

1.4.5.1 Static Random Access Memory (SRAM)

SRAMs (Static Random Access Memory) are volatile synchronous memory components. Their contents do not persist on disk. They can be used to store runtime data for a circuit.

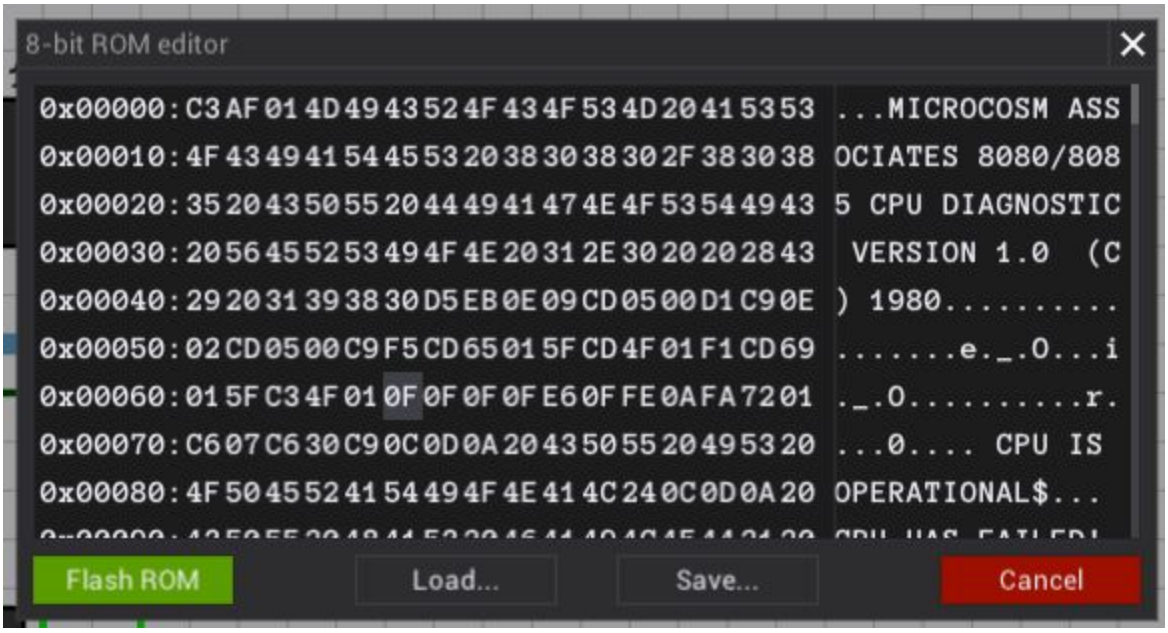
SRAMs have several different pins. Currently, the SRAM component behaves like a **synchronous flow-through standard write SRAM** chip. All inputs are latched on the rising edge of the supplied clock (*clk*) and the addressed memory location is read/written as some later timestep. The read delay of an SRAM component is currently fixed and equal to 5ns.

oe (Output Enable) is an active low signal used to control whether *Dout* will hold the data from the selected address. When *oe* is HIGH, *Dout* is equal to UNDEFINED. *oe* is an asynchronous signal.

cs (Chip Select) is an active low signal used to control whether the rest of the inputs will be latched on the next rising edge of the clock or not.

bwe (Byte Write Enable) is an active low signal used to control whether the addressed memory location will be written on the next rising edge of the clock or not.

1.4.5.2 Read-only Memory (ROM)



ROMs are components used for storing immutable data. Each ROM contains M N-bit words. Words can be up to 64 bits wide and the maximum number of words is 64k.

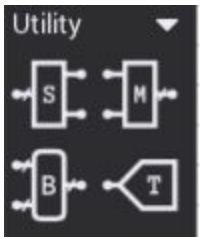
They have 1 input which is the address of the word you want to read and at least 1 output corresponding to the selected word data. In case the word size is larger than 16 bits the component has multiple outputs. From top to bottom each output holds the data from the LSB to the MSB of the selected word.

ROMs are combinational memory components (i.e. there's no clock input). Every time the '*addr*' input changes, the outputs are updated to reflect the newly selected word.

The context menu (right click) of a ROM component has an 'Edit values…' option. Selecting it will show the ROM editor (show above). From there you can change the contents of the ROM, either by typing or by loading an external file. You can also save the current contents of the ROM to a file for later use.

ROM files are saved in the 'roms' folder, under the user's data folder (see section 1.1 for the exact location depending on your OS). If you want to load your own data into a ROM, place your file into the above folder and then load it through the ROM editor. ROM files must have a 'rom' extension.

1.4.6 Utility



The Utility group includes several utility logic components.

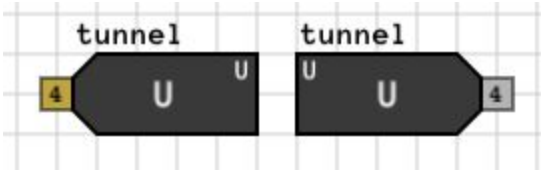
1.4.6.1 Wire mergers and splitters

Wire mergers are used for combining multiple 1-bit signals into a larger signal. Wire splitters, on the other hand, are used for breaking a wide signal into multiple 1-bit signals. The only configurable option for both of these components is the number of bits.

1.4.6.2 Bus

Buses are special components used to select one of many signals. A bus has multiple inputs and only one output, all of them having the same size. The output is always equal to the **one and only non-Undefined input**. If multiple inputs to a bus have a non-Undefined value, the bus output is Undefined. The minimum number of inputs is 2.

1.4.6.3 Tunnels



Tunnels behave like numeric I/O ports. You can only create output tunnels via the gate toolbar, the same way you create numeric output ports. Input tunnels are created via the output tunnel's context menu (right click on it). All input tunnels get the same name as their parent.

Cloning an input tunnel is the same as creating a new input from the same output tunnel. Cloning an output tunnel is the same as creating a new output tunnel via the gate toolbar. Cloned output tunnels have no connection to their original tunnels.

Note that if the current selection includes both an input tunnel and its parent, cloning the selection will create a new output tunnel and a new linked input tunnel. There will be no connection to the original ports.

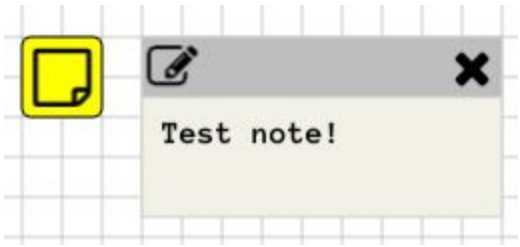
Finally, tunnels do not appear on the final component as pins. They might behave like I/O ports but they aren't actually I/O ports.

1.4.7 Misc

The Misc group includes everything else that doesn't fit in one of the other groups. As of v0.16, Scripted components have been moved to the Misc group

because they are planned to deprecated in future version. They have been replaced by HDL components.

1.4.7.1 Notes

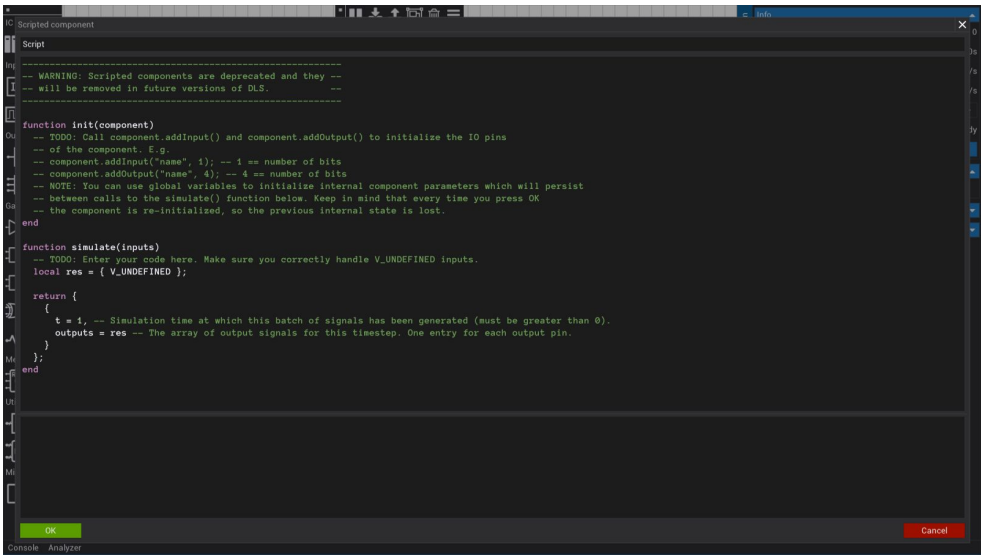


You can add notes to your schematics the same way you add gates and components. Initially, the new note will be closed (yellow symbol). Left clicking on the note will expand it.

When saving your schematics, the state of the note (opened or closed) is preserved so you can keep important notes opened and the rest closed.

Finally, notes are drawn last when rendering the schematic, so they will always appear on top of the circuit (gates/components/wires) independent on the order they were created.

1.4.7.2 Scripted components



The image above shows the scripted component editor. On the top there's a single line edit box for the name of the component. Below it's the editor for the component's code. Clicking OK creates a new component and Cancel closes the dialog. For details on how scripted components work see chapter 4. Note that Scripted components **have been deprecated** on v0.16 in favor of HDL components.

2 Navigation & Wiring

Panning and zooming is done using the mouse. While holding down the right mouse button on an empty part of the schematic, you can move your view around. The mouse wheel can be used to zoom in and out. You can also focus on a specific I/O or Tunnel from the Simulation tab

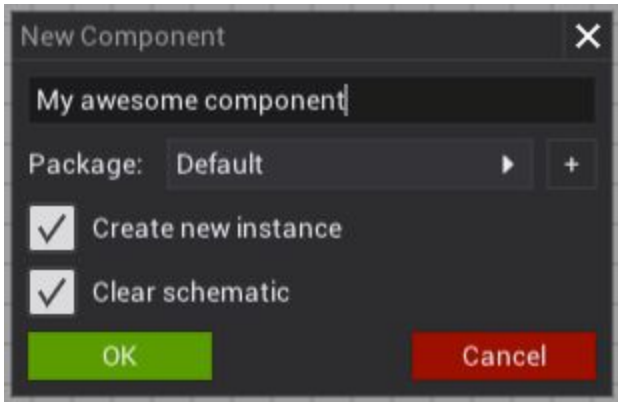
Creating wires is simple. Clicking on any pin will start a new wire. After that, you can place as many points you want on the grid. Finally, click on another pin to finish it.

Using the right mouse button while drawing a new wire removes the last point. If the last point was the first point of the wire (the one on the output pin), the wire is destroyed. Alternatively, you can press Esc to cancel the current wire.

You can select wires by left clicking on them. Right clicking will bring up the context menu from where you can either create a new wire up to this point (T-junction) or delete the selected wires.

3. Components

You can create new components by clicking the “Componentize circuit” button from the Main toolbar. The circuit should have at least 1 input port, 1 output port and 1 wire in order to be turned into a component. Clicking the “Componentize circuit” button will bring up the dialog shown below.



The edit box at the top of the dialog is used as the name of the new component. Next, you can select inside which package you want to add the new component, from a list of available packages. Clicking the + button will open a new dialog from which you can create a new package.

The 2 checkboxes should be self-explanatory. If “Create new instance” is checked, once the OK button is clicked, a new instance of the component will be ready to be added to the circuit. “Clear schematic” clears the current schematic.

The new component will have as input pins all variable input ports (those not marked as Const). Each input port pin (independent of its width) will be shown as an input pin on the new component. The same is true for all non-debug output ports.

Finally, you can inspect the internal circuit of a component by selecting Inspect from its context menu (right click on the component).

4. Scripted components

WARNING: Scripted components have been marked as deprecated in v0.16 in favor of HDL components. This means that they might (or might not) be removed in future versions. Please avoid using them extensively in your circuits until this warning is removed!

Scripted components can be used to create complex behaviours, or in situations where simulation speed matters. The scripting language used in DLS is Lua (Lua 5.3)

Each scripted component should have at least 2 functions specified; `init()` and `simulate()` which are described in more detail below.

4.1 The init() function

This function is used for one-time initialization of the component. It has one argument (named "`component`" in the initial code), which is an temporary Lua object used to specify the component's inputs and outputs.

The "`component`" object has 2 member functions, `addInput()` and `addOutput()`. Both functions take the same number (2) and type of arguments. The first argument should be a string with the name of the pin. The second argument is the width of the pin, in bits.

Script 1 below shows how to create a component with 2 inputs ("A" and "B") and 1 output ("Y"), all 1-bit wide.

```
function init(component)
  component.addInput("A", 1);
  component.addInput("B", 1);
  component.addOutput("Y", 1);
end
```

Script 1: Initializing a scripted component with 2 inputs and 1 output.

The `init()` function is a good place to also initialize internal component state. You can use global variables which will remain valid for the lifetime of the component. Note that the `init()` function is called every time you press the OK button on the editor.

4.2 The simulate() function

The `simulate()` function is where the actual component logic should be placed. Every time an input changes value, the `simulate()` function is called. It has one argument (named "`inputs`" in the initial code), which is an array of numbers, corresponding to each one of the

inputs, in the order they are created in the `init()` function, and should return an array of objects, one for each internal timestep of the simulation.

Inputs can take any valid integer value, depending on the width of the input pin, or the special value of `V_UNDEFINED`.

`V_UNDEFINED` means that either the input pin isn't connected to any output pin from another component/gate/port, or the wire is in high impedance state (hi-Z).

`simulate()` should return an array of objects. Each element in the array corresponds to one internal timestep of the component. Each timestep "object" should have 2 members. One named "`t`" which is the internal timestep value, greater than or equal to 1, and the second named "`outputs`" which should be an array of numbers with the values of each output pin for this timestep, in the order they were created in the `init()` function.

Script 2 shows an example using the inputs and outputs created in Script 1 to model a 1-bit AND2 gate. It also shows the `print()` function which is available to the scripts for printing debug information to the console.

```
function simulate(inputs)
  local a = inputs[1];
  local b = inputs[2];
  local res = { V_UNDEFINED };

  if (a == 0 or b == 0) then
    res[1] = 0;
  elseif (a ~= V_UNDEFINED and b ~= V_UNDEFINED) then
    res[1] = bit.band(a, b);
  end

  return {
    { t = 1, outputs = res }
  };
end
```

Script 2: 1-bit AND2 gate implemented as a scripted component.

5. Testbenches

Testbenches are Lua scripts which can control the inputs and check the outputs of a circuit. They are used to verify if a circuit behaves as it's expected. The table below shows a list of functions available in testbenches, to control the circuit.

Function	**Description**
set(string inputName, number value)	Sets the input named `inputName` to the specified value. Push buttons keep their value until it's set again (like if you were holding the mouse button down). You cannot set the value of a clock. See `tick()` for details. If the input port doesn't exist, the function does nothing.
get(string outputName)	Returns the value of the output port named `outputName`. Only works with Output port and LED Matrix components. If the output port does not exist, the returned value is `V_UNDEFINED`
simulate()	Simulates the circuit using the current input values.
tick(string clockName)	Ticks the clock named `clockName`. One call to this function forces the clock to switch levels, so a full clock cycle is executed after two consecutive calls to `tick()`.
assert(boolean condition, string message)	Check if the condition is true, and if it's not the message is printed to the console, and the testbench terminates.
print(string message)	Prints a message to the console.

Table 1: List of functions available for use in a testbench.

Script 3 below shows a simple testbench for the scripted component we created in the previous section. In order to run it, connect 2 1-bit input ports to the circuit to the component's input pins and an output port to the output pin.

```
set("A", 0);
set("B", 0);
simulate();
```

```
assert(get("Y") == 0, "Invalid output, expected 0");

set("A", 1);
set("B", 1);
simulate();
assert(get("Y") == 1, "Invalid output, expected 1");

set("A", 0);
set("B", V_UNDEFINED);
simulate();
assert(get("Y") == 0, "Invalid output, expected 0");

print("Testbench completed successfully");
```

Script 3: Testbench for the scripted component from scripts 1 and 2.

NOTE: While the testbench is running, you cannot interact with the circuit.

6. Hardware Description Language

From v0.16 and up DLS includes a custom Lua-based HDL. If you are familiar with bitwise's rattle HDL, the syntax is very similar except rattle is written in Python which has some differences compared to Lua.

6.1 Modules

```
require('hdl');

MyMux2 = module[[4-bit Mux2]](
function ()
  i0 = input(bit[4]);
  i1 = input(bit[4]);
  sel = input(bit);

  o = output(when(sel, i1, i0));
end);

top = MyMux2;
```

Listing 6.1: A simple 4-bit 2-input multiplexer

Listing 6.1 above shows the code for a 4-bit 2-input multiplexer. A multiplexer is a component used to select one of many signals based on the value of another signal. Let's look at the code line by line to see what's going on.

`require('hdl')`: Includes the HDL compiler code (not a compiler, but the term should do for this explanation). This line is mandatory and should probably be the first line of each HDL component. If you want to take a look at the code, the file can be found at `<installation dir>/data/hdl`

`MyMux2 = module[[4-bit Mux2]](`: Creates a new `module` named "4-bit Mux2". The `module()` function takes as input a string, which is the name of the generated component and returns a function. The returned function takes as input a function with no arguments which will include the code for your component. The double-bracket notation is a shortcut for `module("4-bit Mux2")(...)`. You are free to use whichever syntax you want. `MyMux2` is the name of the variable which you will use to refer to this module in your code.

`i0 = input(bit[4])`: Creates an 4-bit input. The `input` function takes 1 mandatory argument (the type of the input port) and 1 optional argument (the name of the input

port). If the name is omitted, the input port will be named after the variable it's assigned to. In this case, the generated component will have a 4-bit input port named `i0`. Node types are specified using the `bit` global variable.

`i1 = input(bit[4])`: Creates another 4-bit input port named `i1`.

`sel = input(bit)`: Creates a 1-bit input port named `sel`.

`o = output(when(sel, i1, i0))`: Creates an output port named `o` and assigns the result of the `when()` function to it. `when()` is a build-in function which generates a 2-input multiplexer. For more build-in functions and operators see section 6.4.

`top = MyMux2`: `top` is a global variable expected by the runtime. It should point to the module you want to instantiate.

Another way of building the same multiplexer without using the build-in function `when()` is shown in Listing 6.2 below. It uses tristate buffers and buses to perform the same operation. All components are wired together the same way you would have done it if you did it in the schematic editor.

```
require('hdl');

MyMux2 = module[[4-bit Mux2]](
function ()
  i0 = input(bit[4]);
  i1 = input(bit[4]);
  sel = input(bit);

  case0 = tristate_buffer(i0, ~sel);
  case1 = tristate_buffer(i1, buffer(sel, 1));

  o = output(bus(case0, case1));
end);

top = MyMux2;
```

Listing 6.2: 4-bit 2-input multiplexer using tristate buffers and buses

Inputs are declared the same way as before. `case0` and `case1` are two intermediate nodes holding the two potential outputs of the module. `tristate_buffer()` takes 2 arguments. The first is the input value and the second is the control signal (always 1 bit wide).

`case0 = tristate_buffer(i0, ~sel)`: `case0` will be equal to `i0` if `sel` is equal to 0 and Undefined/Hi-Z if `sel` is equal to 1.

`case1 = tristate_buffer(i1, buffer(sel, 1))`: `case1` will be equal to `i1` if `sel` is equal to 1 and Undefined/Hi-Z if `sel` is equal to 0.

The reason we used a `buffer()` to delay the `sel` signal to the control pin of the tristate buffer is because the unary NOT operator (~) has a propagation delay of 1. If we don't use a buffer, whenever `sel` changes value, there will be a single timestep in the simulation when both inputs to the bus are non-Undefined and as a result the bus output will be Undefined.

`o = output(bus(case0, case1))`: Creates the output `o` and assigns the output of a 2-input bus to it. The `bus()` function creates a bus with the same number of inputs as the number of arguments. Note that all `bus()` arguments should have the same type (in this case `bit[4]`). As a result the `o` output will have a type of `bit[4]`.

6.2 Instantiating modules

In section 6.1 above we saw how to create a new module. In this section we will see how you can instantiate such modules to build larger and more complex components. Listing 6.3 below shows the code for a 4-bit 4-input multiplexer using the 4-bit 2-input multiplexer we saw earlier. You can choose whichever implementation of the 2-input mux from above.

```
require('hdl');

MyMux2 = ... -- Pick one of the 2 implementations from above.

MyMux4 = module[[4-bit Mux4]](
function ()
  q0 = input(bit[4]);
  q1 = input(bit[4]);
  q2 = input(bit[4]);
  q3 = input(bit[4]);
  sel2 = input(bit[2]);

  m01 = MyMux2{i0 = q0, i1 = q1, sel = sel2[1]};
  m23 = MyMux2{i0 = q2, i1 = q3, sel = sel2[1]};

  m0123 = MyMux2();
  m0123.i0 = m01.o;
  m0123.i1 = m23.o;
  m0123.sel = sel2[2];

  o = output(m0123.o);
end);
```

```
top = MyMux4;
```

Listing 6.3: 4-bit 4-input multiplexer using MyMux2 module from section 6.1

Inputs are declared the same way as before. Since this is a 4-input mux we need 4x 4-bit inputs. The `sel2` input is now 2 bits wide because we have 4 different values to select from. The names of the input ports for this module has been changed from `i` to `q` to avoid any confusion when connecting them to the various modules. You are free to pick any name you want for them.

`m01 = MyMux2{i0 = q0, i1 = q1, sel = sel2[1]}`: Creates a `MyMux2` instance and connects `i0`, `i1` and `sel` to `q0`, `q1` and `sel2[1]` respectively.

Modules are Lua functions (technically they are callable tables but that doesn't changed the following description). In order to create an instance of a previously defined module you have to call its corresponding function. In this case the `MyMux2()` function is called with a single Lua table as its argument. The above notation is shorthand for `MyMux2({...})`. Lua doesn't support named arguments and this is the only way to assign values to a module's inputs when creating an instance.

Another part of the above line worth noting is the way a single bit is extracted from the 2-bit input `sel2`. You use the index operator ([]) on the multi-bit node to extract a single bit out of it. Since the `sel` input of `MyMux2` is 1 bit wide, we cannot connect `sel2` directly to it. We use the least significant bit of `sel2` to pick one of `q0` and `q1`.

Finally note that Lua indices start at 1, so the least significant bit of a node is at index 1 and the most significant bit is at index `size`.

`m23 = MyMux2{i0 = q2, i1 = q3, sel = sel2[1]}`: Creates a second instance of `MyMux2` to select between `q2` and `q3` based again on `sel2[1]`.

`m0123 = MyMux2()`: Creates a third instance of `MyMux2` but doesn't assign any values to its inputs. You are free to assign as many inputs to a module instance during instantiation and leave the rest for later.

`m0123.i0 = m01.o`: Connects the output `o` of `m01` to the `i0` input of `m0123`.

`m0123.i1 = m23.o`: Connects the output `o` of `m23` to the `i1` input of `m0123`.

`m0123.sel = sel2[2]`: Connects the most significant bit of `sel2` to the `sel` input of `m0123`.

`o = output(m0123.o)`: Creates an output named `o` and assigns the output `o` of `m0123` to it.

6.3 Slices and concatenations

Above we saw how you can select individual bits out of node using the index operator []. Sometimes you might want to select a range of bits from a node. There are currently 3 ways to achieve this and they are all equivalent. It's just a matter of preference which one you'll choose. Unfortunately, Lua doesn't have the notion of array slices like Python, that's why we've implemented 3 different ways for you to pick.

You might also want to concatenate multiple nodes together to form a wider signal. Listing 6.4 shows some examples.

```
require('hdl');

SliceAndConcat = module[[SliceAndConcat]](
function ()
  i = input(bit[8]);

  q12 = i(1,3);        -- call i as a function
  q23 = i[slice(2,4)]; -- indexing with a slice()
  q57 = i:slice(5,8);  -- call the slice() method of i

  q1257 = q12 .. q57; -- concatenate q12 and q57 into a 5-bit
signal

  o1 = output(q23);
  o2 = output(q1257);
end);

top = SliceAndConcat;
```

Listing 6.4: Examples of slicing and concatenation

All three versions of slicing require 2 integer arguments: the start and end of the slice. The end bit is not included in the slice. For example, `i(1,3)` will extracts the 1st and 2nd bit of `i` and create the 2-bit node `q12`. Remember that Lua indices start at 1. Similarly `q57` is 3 bits wide (bit at index 8 is excluded).

Concatenation is performed with the Lua concatenation operator (..).

6.4 Build-in components and operators

Function	Description	Example

`input(type, name)`	Creates an input port. *name* parameter is optional. If not specified, the name of the input port will be the name of the variable it's assigned to.	`i0 = input(bit)`
		`i1 = input(bit, "name")`
		`i2 = input(bit[4])`

`output(node)`	Creates an output port and assigns the *node* as its source.	`o = output(~i0)`
Unary operator ~	Generates a NOT gate.	`_not = ~i0`
Binary operator &	Generates an AND gate.	`_and = i0 & i1`
Binary operator |	Generates an OR gate.	`_or = i0 | i1`
Binary operator ~	Generates a XOR gate.	`_xor = i0 ~ i1`
Binary operator >>	Generates a Shift Right component. Shift amount must be a constant.	`_shr = i2 >> 2`
Binary operator <<	Generates a Shift Left component. Shift amount must be a constant.	`_shl = i2 << 2`
Binary operator ..	Concatenation of 2 nodes. Result node size is equal to the total size of the 2 nodes.	`_concat = i0 .. i1`
`const(type, value)`	Creates a constant input port with the specified *type* and *value*.	`c = const(bit[2], 2)`

`register(type, clk, rst_n, set_n, next)`	Creates a register. *clk* is the clock and it's the only mandatory input pin. *rst_n* and *set_n* are 1-bit active low inputs for resetting and setting the register. If they aren't specified they are automatically connected to `const(bit,1)`. *next* is the next value that will assigned to the register on the rising edge of the clock input.	`r = register(bit, clk)`
		`r.next = ~i1`
		`r.rst_n = i0`

`eq(node,node)`	Creates a comparator.	`cmp = eq(i0,i1)`

`ne(node, node)`
`le(node, node)`
`lt(node, node)`
`ge(node, node)`
`gt(node, node)`		

`bus(node, node, …)`	Create a bus.	`_bus = bus(i0, i1)`

`mux2(c0, c1, sel)`
 	
`mux4(c0, c1, c2, c3, sel)`
 	
`mux8(c0, c1, c2, c3, c4, c5, c6, c7, sel)`
	Multiplexers.
	sel should be the correct size for each multiplexer: 1-bit for mux2, 2-bit for mux4 and 3-bit for mux8.
		`res = mux2(i0, i1, sel)`

`muxn(cases, sel)`	Generic multiplexer. *cases* should be an array of signals/nodes. *sel* restrictions apply.	`res = muxn({i0,i1),sel)`
`when(cond, then, else)`	Shortcut for mux2()	`res = when(sel, i1, i0)`
`buffer(node, delay)`	Creates a Buffer with the specified delay.	`buf = buffer(i0, 5)`
`tristate_buffer(node, control)`	Creates a Tristate Buffer.	`ts = tristate_buffer(i2, i0)`
`pull(node, value)`	Creates a Pull Resistor.	`res = pull(i2, 0)`
`Index operator []`	Extracts the bit at the specified index.	`res = i2[1]`

`Slicing`	Extracts multiple consecutive bits from the node	`res = i2(1,3)`
		`res = i2[slice(1,3)]`
		`res = i2:slice(1,3)`

`bit_length(x)`	Returns the minimum number of bits required to represent the integer *x*.	`len = bit_length(5)` `-- will return 3`
`switch(reg)/case(x)/default()`	See section 6.6	
`assign(reg, node)`	Assign *node* to the next value of	

	reg. When used outside of switch()/case() blocks it works as a shortcut to *reg.next* = *node*.	

6.5 Custom Adders

There's no build-in adder module in the HDL but the + and - operators are supported. You just have to register your own function to be used as the adder circuit. This is done by calling `hdlRegisterAdder(func)` immediately after `require`()'ing the hdl.

`hdlRegisterAdder()` takes as input a function. This function should have 3 arguments (`x`, `y` and `carry in`) and produce 2 results (`sum` and `carry out`). Listing 6.5 shows an example of such an adder. You can put this file in your HDL library under whatever name you choose and `require()` it in your modules.

```
require('hdl');

-- Full adder
FullAdder = module[[FullAdder]](
function()
  x = input(bit);
  y = input(bit);
  ci = input(bit);
  p = x ~ y;
  g = x & y;
  s = output(p ~ ci);
  co = output(g | (p & ci));
end);

-- Wrapper function for the FullAdder module
function add3(x, y, c)
  local adder = FullAdder{x=x, y=y, ci=c};
  return adder.s, adder.co;
end

-- N-bit ripple carry adder.
function adc(x, y, c)
  c = c or 0;
  local s = {};
  local si;
  for xi, yi in _zip(x, y) do
    si, c = add3(xi, yi, c);
```

```
    s[#s + 1] = si;
  end
  return bits(s), c;
end

-- Helper function:
-- https://stackoverflow.com/a/36096338
function _zip(...)
  local arrays, ans = {...}, {};
  local index = 0;
  return function()
    index = index + 1;
    for i,t in ipairs(arrays) do
      if(type(t) == 'function') then
        ans[i] = t();
      elseif(type(t) == "number" and math.type(t) == "integer")
then
        ans[i] = (t >> (index - 1)) & 1;
      else
        ans[i] = t[index];
      end

      if(ans[i] == nil) then
        return;
      end
    end

    return table.unpack(ans);
  end
end
```

Listing 6.5: A simple ripple carry adder

Listing 6.6 below shows how such an adder can be used for the + and - operators. The generic N-bit adder function from Listing 6.5 is `adc()` which is the function we register as the adder module.

```
require('hdl');
require('adder'); -- Code from listing 6.5

hdlRegisterAdder(adc);
```

```
TestAddSub = module[[Test Add/Sub]](
function ()
  x = input(bit[4]);
  y = input(bit[4]);

  x_plus_y = x + y;
  x_minus_y = x - y;

  o_add = output(x_plus_y);
  o_sub = output(x_minus_y);
  o_add_carry = output(x_plus_y[5]);
end);

top = TestAddSub;
```

Listing 6.6: Using the adder module from listing 6.5

The + operator calls your registered adder as `adder(x, y, 0)` and the - operator calls the registered adder as `adder(x, ~y, 1)`.

Finally, note that the carry out can be accessed by reading the (N+1)-th bit from the result of the addition. In the above example N = 4 so the carry out is located at index 5 of `x_plus_y`.

6.6 switch() blocks

`switch()` blocks can be used to assign values to multiple **registers** based on the value of another signal. They are useful for creating state machines. Instead of using multiplexers for each individual register, you `assign()` all registers on the same `case()` block.

Since this is actually Lua code and Lua doesn't support `switch()` statements, this is implemented using functions. The single-parameter function-call syntax of Lua helps make the code look like regular `switch()` blocks but there are some important caveats. Let's look at an example first.

```
require('hdl');

TestFSM = module[[Test FSM]](
function ()
  local STATE0 = const(bit[2], 0);
  local STATE1 = const(bit[2], 1);
  local STATE2 = const(bit[2], 2);
```

```
  local STATE3 = const(bit[2], 3);

  clk = input(bit);
  en = input(bit);

  state = register(bit[2], clk);
  done = register(bit, clk);

  switch(state) {
    case(STATE0) {
      assign(state, when(en, STATE1, STATE0)),
      assign(done, 0);
    },
    case(STATE1) {
      assign(state, STATE2)
      -- NOTE: done doesn't get assigned here. It will keep its
previous value.
    },
    case(STATE2) {
      assign(state, STATE3)
    },
    case(STATE3) {
      assign(state, STATE0),
      assign(done, 1)
    },
    default() {
      assign(state, STATE0),
      assign(done, 0)
    }
  };

  o_done = output(done);
  o_ready = output(eq(state, STATE0));
end);

top = TestFSM;
```

Listing 6.7: switch()/case() example

Listing 6.7 shows a simple state machine. The next `state` and `done` values are calculated based on the current value of `state` and the `en` input. If the current `state` is

equal to `STATE0` and `en` is high, the state machine switches to `STATE1`, `STATE2`, `STATE3` and back to `STATE0` in turn, whenever `clk` goes from low to high.

Technically, `switch()` is a function which returns a function. The returned function takes only a single argument, that's why we can omit the parenthesis when invoking it. The argument should be a table of `case()` calls.

`case()` is also a function which returns a function. Each `case()` call generates an equality comparator between the signal/node passed to it and signal/node passed to the parent `switch()` call. The function returned by the `case()` call takes a single table of `assign()` calls as argument.

`assign()` takes two arguments. The first argument should be a register and the second a signal/node which will assigned to the register on the next clock tick.

If the above look a bit complicated the thing to remember is that you **must** use `assign()` inside `case()/default()` blocks and that all those "blocks" are actually Lua tables, so commas between statements is required. You cannot put regular Lua statements (e.g. `state = STATE2`) inside a case block. You have to use `assign()`.

A. Build-in Gate/Component Information

Table A.1 below shows the delay of each one of the basic gates based on its number of inputs. The number of bits isn't taken into account because each individual bit is processed in parallel to the rest.

Currently (as of version 0.11.0) 1 gate delay equals to 1 nanosecond, which is also the smallest amount of time a circuit can advance forward in time. As a result, the maximum effective clock frequency is 500MHz (2 gate delays are required for a complete clock cycle).

	Number of Inputs								
Gate	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**N**
Buffer	var	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
NOT	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AND	N/A	1	2	2	3	3	3	3	log2(nextPowOf2(N))
OR	N/A	1	2	2	3	3	3	3	log2(nextPowOf2(N))
NAND	N/A	1	2	2	3	3	3	3	log2(nextPowOf2(N))
NOR	N/A	1	2	2	3	3	3	3	log2(nextPowOf2(N))
XOR	N/A	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Tristate Buffer	N/A	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Pull Resistor	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table A.1: Basic logic gate delays (in arbitrary time units)

Table A.2 shows the delays of the rest of the build-ing components.

Component	**Delay**
Bus	1
Wire Merger	1
Wire Splitter	1
ROM	Number of Address bits

Table A.2: Delays for the components from the I/O dialog.

B. Release History

v0.16.3 (2019-01-27)

1. NEW: HDL: switch()/case() blocks.
2. NEW: HDL: Added muxn() and bit_length() functions.
3. FIX: Crash on macos with Metal renderer when creating pixel displays.
4. FIX: Changed propagation delays of HDL-only simulation components.
5. FIX: Crash when calling tick() in testbenches without a clock name.
6. FIX: HDL: Register's default value after creation is now itself.
7. FIX: Clock frequency in clock configuration dialogs is now displayed properly.
8. FIX: Logic Analyzer: Center the value of multi-bit signals inside their box even if the box is larger than the screen.

v0.16.2 (2018-12-29)

1. NEW: User-defined adder modules for the + and - operators.
2. NEW: HDL: Slightly better error reporting with stack trace.
3. FIX: Crash when starting the 1st level on the first run.
4. FIX: Changed Linux user data folder location to avoid locale specific Documents folder.
5. FIX: HDL: Comparator nodes had the wrong width.
6. FIX: HDL: slice() global function was missing.

v0.16.1 (2018-12-22)

1. NEW: Custom Lua-based Hardware Description Language (HDL)
2. NEW: New UI and dark theme
3. NEW: New schematic graphics
4. NEW: Logic simulator rewrite
5. NEW: New and exciting bugs for you to discover
6. FIX: Probably a lot of bugs (didn't keep detailed logs because a lot of the code was rewritten).

v0.15.0 (2017-07-21)

1. NEW: Pull up/down resistors
2. NEW: Logic Analyzer enhancements
3. NEW: Copy component schematic command in component's context menu
4. NEW: Better gate/component toolbar
5. FIX: SRAM output timings on write were wrong.
6. FIX: Schematic file format has been changed slightly to save IO pins before everything else.

v0.14.0 (2017-04-11)

1. NEW: Notes
2. NEW: Faster renderer

3. NEW: Faster simulator
4. NEW: Removed error values. V_ERROR is no longer defined in scripted components.
5. FIX: Clicking on read-only edit boxes was crashing the app.
6. FIX: Connecting the output of a gate/component to an input of the same gate/component is forbidden (was causing problems with wires).
7. FIX: Random crash related to constant input ports.

v0.13.0 (2017-02-03)

1. NEW: Undo system
2. NEW: Signal tunnels to connect distant parts of a circuit without making a mess of wires.
3. NEW: Static RAM component (synchronous, flow-through, standard write SRAM)
4. NEW: Configurable LED color
5. NEW: Wire thickness based on number of bits.
6. NEW: Edit buffer propagation delay
7. NEW: Increased maximum unzoom level and made the grid adaptive (to avoid rendering too many lines at small zoom levels)
8. NEW: Increased maximum target simulation speed to 1s/s.
9. FIX: Buffers didn't check for UNDEFINED and ERROR values
10. FIX: Component bounding boxes were wrong (didn't take into account constant/debug I/O ports).
11. FIX: Large range slider change speed is now dependent on the range.
12. FIX: Large range slider +/- labels turned into buttons to quickly change the current value.
13. FIX: Giving a name to a new component and then opening the new package dialog was crashing the app.
14. FIX: When returning to the parent schematic the previous view is restored.
15. FIX: Vertical scrollbar in listboxes with N+1 elements, where N is the number of visible elements.

v0.12.0 (2016-12-21)

1. NEW: Select multiple components and wires by dragging the mouse on the grid, or by holding down the Shift key.
2. NEW: Rotate gates and some of the build-in components using the mouse wheel.
3. NEW: Right clicking on a wire brings up its context menu (doesn’t remove the wire anymore).
4. NEW: New Buffer gate with configurable delay (up to 50ns).
5. NEW: Gate/component toolbar has been moved to the left of the screen and all build-in components and gates are displayed. No more I/O and misc gates dialog.
6. NEW: Toolbars can now be unpinned and automatically fold when the cursor isn’t near them.
7. NEW: Reduced the size of the schematic file format. Now each component in a schematic is saved only once, independent of the number of instances in the schematic.
8. FIX: Fixed the bounding boxes of some gates/components.

9. FIX: Display textures are now aligned correctly inside the component rectangle for all resolutions.
10. FIX: Stream levels with 0-terminated sequences have random values but the sequence lengths are fixed.

v0.11.2 (2016-11-27)

1. NEW: Added the total number of transistors in the current circuit to the Simulation/Circuit info panel.
2. FIX: Fixed the description of the Tri-state buffer tutorial level about buses and Undefined values.
3. FIX: When the Simulation/Circuit info panel was visible the Escape key didn't work (didn't bring up the Pause menu).

v0.11.1 (2016-11-19)

1. FIX: When adding or removing wires from inputs to components/gates, the gate's input pin wasn't set to the correct value.
2. FIX: Truth table levels didn't handle signed I/O correctly during verification.
3. FIX: If you created a new component in the sandbox and then started a level to use the component, the component wasn't included in the package. You had to restart the game.

v0.11.0 (2016-10-26)

1. NEW: Simulator internals have changed. Circuits are now flattened to their basic logic gates (instead of working on a tree of components) and simulation time is advanced forward if there are any events queued. Steady-state isn't required in order to finish simulating a circuit, so both unstable (i.e. oscillators) and asynchronous circuits should work correctly.
2. NEW: Basic logic analyzer. Every debug output is monitored and displayed in the analyzer.
3. NEW: You can now clear Display components to black or random color via their context menu.
4. FIX: Histogram bucket size was wrong and extreme values weren't displayed properly.
5. FIX: Multi-bit OR gate gave wrong results when one of its inputs was equal to 1.
6. FIX: Inspecting a component while the parent circuit had a testbench, caused the testbench to reset when returning to parent.
7. FIX: During level editing, when the user created a new component and then inspected another component, when returning to the new component's schematic the main menu toolbar was wrong.

v0.10.1 (2016-09-28)

1. NEW: Level statistics. Compare your results against other players!
2. NEW: Added a red border around the schematic editor which is displayed only if the last circuit simulation was unstable.

3. NEW: Credits dialog
4. FIX: Window icon (Windows + Linux)
5. FIX: Fixed tooptip location when it got out of screen.
6. FIX: Package names were getting mixed up when deleting packages from the library.
7. FIX: Background image was not rendered correctly with UI scale other than 1.0, in pause menus
8. FIX: Editing the schematic after an unstable simulation was crashing the app.

v0.10.0 (2016-09-16)

1. NEW: The first 37 levels!
2. NEW: Respect user's keyboard layout (e.g. QWERTY, Dvorak, AZERTY) when writing into text fields. At the moment all shortcuts (Ctrl + C/V/X) require pressing the corresponding QWERTY key.
3. NEW: Buses and tristate buffers (TSB) do not produce error values any more. When the control input of a TSB is UNDEFINED, its output is UNDEFINED. Buses forward the first non-UNDEFINED value to the output (from top to bottom).
4. NEW: New main menu UI.
5. NEW: You can cancel wire creation by pressing the ESC key.
6. FIX: Fixed a bug which appeared when undoing wire point placement. If the mouse moved a bit while clicking the right mouse button, the operation wasn't executed.
7. FIX: When deleting gates/components without any wires attached, the confirmation dialog isn't displayed.
8. FIX: You can now change clock values using the mouse, just like regular numeric inputs, when the clock isn't ticking.
9. FIX: Fixed multi-input gate delays. The more the inputs the larger the delay. See Appendix A from the manual for more details.
10. FIX: If there's no string in the clipboard the app was crashing when pasting into an text field.
11. FIX: Log file wasn't sent along with the crash dump (Windows only)

v0.9.0 (2016-07-16)

1. NEW: ROMs can have word size up to 64 bits.
2. NEW: Multi-bit multi-input standard gates.
3. NEW: Load/save ROM data from/to external files.
4. NEW: 7-segment displays
5. FIX: Changing any window related option which caused bgfx to the reset, ended up crashing the game if there was previously a display in the circuit and a display was added after the reset.

v0.8.1 (2016-04-28)

1. NEW: Added a small manual describing the sandbox (this document).
2. FIX: Calling internal DLS functions from Lua scripts ended up crashing the application if the parameter types weren't correct.

3. FIX: Converting input ports was causing a crash when the port had more than 1 wire.
4. FIX: No 6 from previous release didn't behave correctly with testbenches.
5. FIX: Components in packages weren't sorted correctly.
6. FIX: Testbenches are not correctly saved within components.
7. FIX: New ROMs are now reset to 0.
8. FIX: Push button Debug parameter renamed to Const.

v0.8.0 (2016-04-13)

1. NEW: Increased the max. number of bits per wire/pin to 16, and the max. number of ROM bytes to 64k.
2. FIX: Minimizing and restoring the window was causing the V-Sync flag to be reset.
3. FIX: ROM editor window contents were getting out of bounds.
4. FIX: When returning to a parent schematic, sometimes the schematic wasn't centered correctly on screen.
5. FIX: Disabled auto UI scale by default to prevent bugs with wrong physical screen dimensions reported by the OS.
6. FIX: When adding or removing wires (directly or indirectly), the circuit is forced to be simulated to prevent undefined values.

v0.7.1 (2016-03-14)

1. FIX: Deleting components from the library caused a crash.
2. FIX: Printing too many messages to the console caused a crash.

v0.7.0 (2016-03-13)

1. NEW: Added testbenches (1 per schematic). You can open the testbench editor by pressing F2. Testbenches are saved along with the schematic (.sch v1.4).
2. NEW: Replaced Duktape with LuaJIT (Lua 5.1 with BitOp library). Unfortunately, your previous circuits written in JS will no longer load/work.
3. NEW: Added V-Sync option in the Options dialog.
4. FIX: Console now opens with the F1 key.
5. FIX: Windows: Schematics and packages are now saved under your Documents\DLS folder (%USERPROFILE%\Documents\DLS). You can move your old schematics/packages from %APPDATA%\DLS into this folder.
6. FIX: bgfx now uses the appropriate renderer for each platform (DX under Windows, OGL under Linux/OSX).
7. FIX: Fixed a bug in the renderer which occured when the number of vertices per frame exceeded 65536.
8. FIX: OSX: Auto UI scale is calculated based on the physical monitor dimensions returned by GLFW.
9. FIX: Several UI fixes and improvements.

v0.6.1 (2016-01-24)

1. NEW: Uses bgfx for handling rendering details on all platforms. Currently only the OpenGL backend of bgfx is used on all 3 platforms, but this might change later (e.g. DX backend on Windows and Metal backend on OSX).
2. NEW: OSX and Linux builds. The Linux build has been tested on Ubuntu and it's expected to work on any Debian based distro (don't take my word on that :)). The OSX build has been tested on El Capitan. If the game is in alpha stage, these 2 builds should be considered pre-alpha at the moment. Please report any bugs either on the itch.io page or twitter (@jdryg).

v0.6.0 (2016-01-15)

1. NEW: Switches and push buttons.
2. NEW: LED matrix display.
3. NEW: 1-bit per pixel display with internal memory (resolutions from 32x32 up to 800x800).
4. NEW: Adjustable clock frequencies (from 1 up to 60Hz). A special value of 0 indicates that the clock will tick as fast as possible (normally at 60Hz with VSync enabled).
5. NEW: (EXPERIMENTAL) Scripted components. You can now build custom components with JavaScript. Duktape JS engine is used internally.
6. NEW: Tri-state buffers and buses.
7. NEW: Conversions between the various input and outputs port from the context menu.
8. NEW: In-game console for tweaking and monitoring of scripted components (via print() calls). You can open the console using the '`' button.
9. FIX: Editbox fixes (numpad support, mouse selection and caret placement, etc.)
10. FIX: Added a read-only editbox to the top of the save/load schematic dialogs which shows the folder used to store the schematics.
11. FIX: Various other bug fixes and improvements.

v0.5.0 (2015-12-10)

1. NEW: New and improved UI.
2. NEW: Inline editing of input port values.
3. NEW: Debug output ports. They function as normal output ports but no output pin is added in the generated component.
4. NEW: Convert between 1-bit input ports and clock.
5. NEW: Convert between const/variable input and normal/debug output ports.
6. NEW: Added a "delete package" button in the library window to remove unwanted packages.
7. FIX: Various bug fixes and memory improvements.

v0.4.2 (2015-11-23)

1. FIX: The simulator gave wrong results/outputs for certail circuits with large delays.
2. FIX: If you cleared the schematic while drawing a new wire (e.g. by clicking on the Clear Schematic button on the main toolbar), the game crashed.

v0.4.1 (2015-11-15)

1. FIX: Const inputs were incorrectly visible in the generated component.
2. FIX: Switching packages from the Library dialog sometimes caused a crash.
3. FIX: Combobox item height didn't match listbox item height (wrong selection at the edges of comboboxes).

v0.4.0 (2015-11-14)

1. NEW: Rotate gates and bus splitters/mergers using the R key.
2. NEW: Added a ROM component (max size: 8 data bits x 8 address bits = 256 bytes).
3. FIX: Deleting schematics via the Load Schematic dialog has been fixed.
4. FIX: Automatic zoom has been limited. When loading circuits with (e.g.) only 1 gate, the zoom factor got too big.
5. FIX: You can now zoom with the mouse wheel if the cursor is on a gate, if the gate doesn't consume the mouse wheel scroll (i.e. inputs ports)
6. FIX: Fixed scrollbar heights to correctly show the scrolling region.
7. FIX: When deleting a component from a package, the preview is removed.
8. FIX: Schematic filenames are sanitized before saving the schematic in order to remove invalid filename characters. All invalid characters are replaced with dashes.

v0.3.1 (2015-11-09)

1. FIX: Fixed a bug which appeared when deleting an input gate when it was already connected to some other gates, and then simulating the circuit.

v0.3.0 (2015-11-08)

1. NEW: Switched to binary files for schematics and packages. The game still supports the old file format, so your old schematics are still valid. Packages get automatically converted to the new format, on the first run.
2. NEW: Changed Edit to Inspect in component's context menu. From now on, in order to change the inspected component's circuit, you have to componentize/save the inspected circuit, change it and replace it in the parent schematic, manually. This has been done in order to simplify inspection.
3. NEW: You can now change input port values with the mouse wheel, while the cursor is on the value. This helps when working with inputs with more than 4 bits.
4. NEW: Ctrl+dragging a gate/component clones the component.
5. FIX: When loading or inspecting a schematic/component, the view automatically zooms to fit the new schematic.
6. FIX: Removed a bug which allowed the user to rename a gate/component to an empty name.
7. FIX: When switching packages in the library dialog, the previously selected component gets unselected.
8. FIX: Zooming in/out with the cursor is performed only if the cursor isn't on top of a component/gate. This was required for change no. 4 above.
9. FIX: When inspecting a component, the new schematic is loaded asynchronously.

10. FIX: Several performance and memory optimizations. Loading times and memory requirements have been improved.

v0.2.0 (2015-10-31)

1. NEW: Added UI scale option. You can scale the UI from 50% up to 300% in case it's too small on your monitor (e.g. hdpi displays). The automatic algorithm tries to calculate the monitor's DPI and scale the UI to match the dimensions of a 96dpi display.
2. NEW: Schematic loading has been made asynchronous.
3. NEW: Pan/zoom using the keyboard. Use WSAD or the arrow keys to pan, Q/Z or numpad +/- to zoom.
4. NEW: Added tooltips to gate toolbar buttons. If you hover over a button, you get information about the specific gate.
5. NEW: You can now start creating a new wire by holding down the Ctrl key and clicking on an existing wires.
6. NEW: Added a Delete button in the Load Schematic dialog which lets you delete previously saved schematics.
7. NEW: When popping a circuit from the stack, a confirmation dialog has been added to let you choose whether you want to replace the old IC with the new one, or revert all possible changes. This message is displayed even if you didn't touch the child circuit. This will fixed in a future version.
8. FIX: Completely disable interaction with the schematic, if a modal dialog is open.
9. FIX: Bus wires are now green when all their bits are HIGH, and red when all their bits are LOW. Intermediate values are displayed as orange.
10. FIX: Loading a previously saved schematic now correctly loads input values.
11. FIX: Changed bus splitter/merger gate size to align with the grid.
12. FIX: There was a point near the top left of the screen where all input/output pins of the gates in the toolbar became hot. If you clicked on it, the game crashed because those gates aren't part of a schematic.
13. FIX: When adding a new wire to a gate input pin, the previously connected wire (if any) is deleted.
14. FIX: When clicking on an output pin which already has wires, none of the existing wires is selected.

v0.1.0 (2015-10-28)

1. Initial release