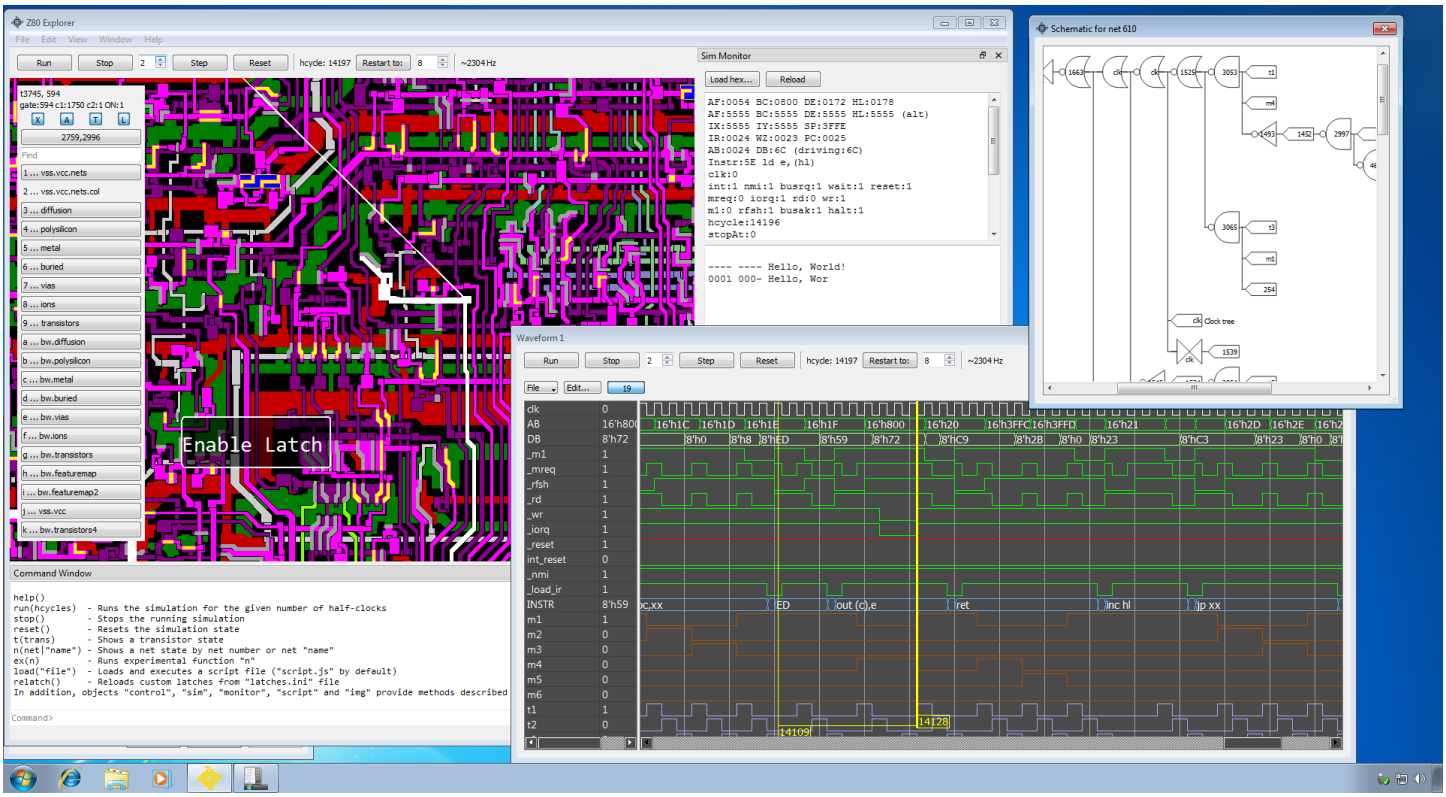
Z80 Explorer

A Z80 Netlist-level Simulator



https://baltazarstudios.com/z80explorer

Updated on 2026-1-06

For the version (1.09)

Contents

Overview

Z80 Explorer is a Z80 netlist-level simulator capable of running Z80 machine code. Its goal is to be an educational tool with features that help reverse engineer and understand this legendary CPU better.

Installation

This application does not need installation. Extract it to a folder of your choice and run it.

Main Application / Image View

The main application window shows a view of the original NMOS Z80 chip die through various layers. This view lets you see various chip features like nets, transistors, and vias. You can combine those layers into a composite view. It is like an X-ray of the chip die.

The main application recognizes these keyboard hotkeys:

Main window keyboard assignments	
Ctrl + Q	Open a new Image View window (up to 4)
Ctrl + W	Open a new Waveform View window (up to 4)
F2	Edit net names (rename and delete names)
F3	Edit definitions of buses (collection of nets)
F4	Edit Watchlist (list of nets with history data)
F5	Edit custom image annotations
F6	Edit custom nets colors
F10	Show or hide the Application Log window
F11	Show or hide the Command Window
F12	Show or hide the Sim Monitor window

These keyboard hotkeys relate to the image view:

Image view keyboard assignments	
1-9, a-k	Select one of the image layers
CTRL+ 1-9,a-k	Adds (composite XOR) selected image on top of the previous one(s)
F1	Cycle zoom modes: Fill, Fit, Identity (1:1), Scale
Cursor arrows	Pan up/down/left/right (also use mouse to pan)
PgUp/PgDown	Zoom (also use the mouse wheel to zoom)
X	Show or hide active nets
SPACE	Show or hide annotations
T	Show or hide active transistors
L	Show or hide detected and custom latches
ESC	Progressively clear Find net, Selected nets
N	Show of hide net names (visible when zoomed in)
SHIFT + X	When active nets are shown, toggle drawing them as full outlines vs. segmented polygons. The latter method helps visualize the nets crossing into a different layer. You will usually see a via or a buried contact connecting the two parts.
Z	When active nets are shown, toggle drawing them in reversed order. Depending on the drawing order, different nets that are crossing, or running in parallel on a different layer, may be hidden. Using this option, hidden nets can be shown above the others.
SHIFT + Z	Adds to the option above (“Z”) to auto-toggle the drawing order every half a second.
, (comma)	(1.07) When nets are shown, cycle drawing them in several ways (see below) Holding the CTRL key reverts to a previous mode.
. (period)	Cycle through the transistor view modes

There are four transistor view modes that help find the operational transistors and latches related to events or instructions.

1. “Active” shows transistors that are currently active (open, pass-through, or ON)
2. “Single-Flip” shows the transistors that have changed the state only once
3. “Sticky” shows all transistors that have changed the state one or more times
4. “All” simply shows every transistor

Press the period key (“.”) to cycle the mode.

For “Single-Flip” and “Sticky” modes, set the base state by pressing the “T” key to toggle the overall transistor view off and back on. After doing that, the internal counters that keep the track of how many times each transistor has changed state will be reset.

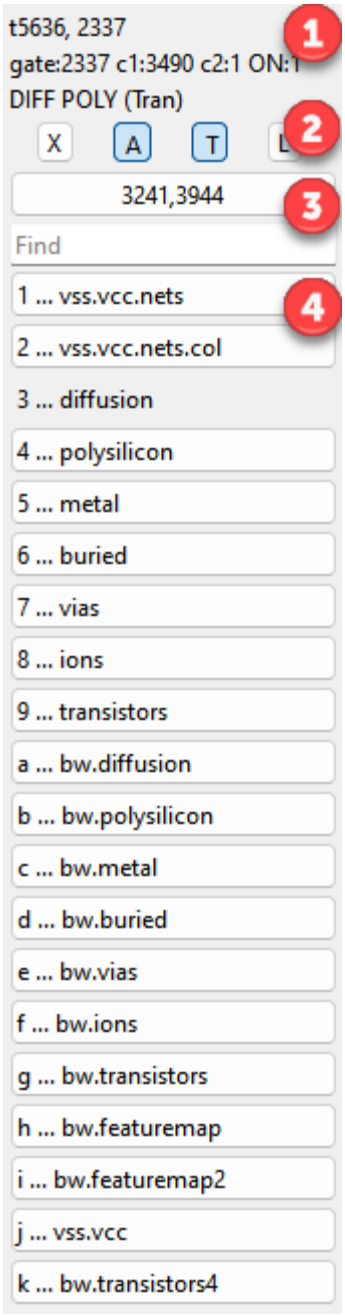
Those two views track transistor changes only when stepping the cycles, or at the end of each run.

(1.07) When nets are shown, pressing the comma key cycles drawing them in several modes. Note that all modes except the "Active" mode are "static", meaning they will not change the state if you step the simulation. Those modes are designed to help you see the topology of those classes of nets:

1. “Active” shows nets that are currently active; this is the default view
2. “Pull-up (static)” shows nets that have pull-ups attached
3. “Gate-less (static)” shows nets that do not drive any transistor gates
4. “Gate-less no Pull-up (static)” shows nets that do not drive any transistor gates and have no pull-ups attached

Nets with no pull-ups are sometimes used to keep the charge for half a cycle or more.

On the left side of the image view is the overlay consisting of several sections:



(1) As you move the mouse cursor over the chip image, this section shows some information about the current net and the chip layers under the cursor.
Also, any currently active net and transistor (logical "1") has its name/number in **bold**.
(2) Four toggle buttons show or hide features on the image (see below).
(3) This button shows the mouse coordinates on the image. Click on the it to reposition the view to another location on the image.
Under it, use the "Find" edit box to search the nets, symbols, or transistors.
(1.07) Find has an autocomplete: as you start typing, it will offer suggestions from the list of known nets. Use up/down keys to select.
(4) This is the list of all available chip layers. Select or click to show a layer or combine the layer images by pressing the corresponding key(s) while holding down the CTRL key.

For example, if you want to see the diffusion and poly layers along with their buried contacts, press "3", then hold CTRL while pressing "4", "6".

Layer "2" is identical to layer "1" but has nets and bus coloring applied. The coloring is described in the section below.

Many layers are duplicated in black/white, and those are prefixed by "bw." Some features are more pronounced when merging the monochrome images into a desired view.

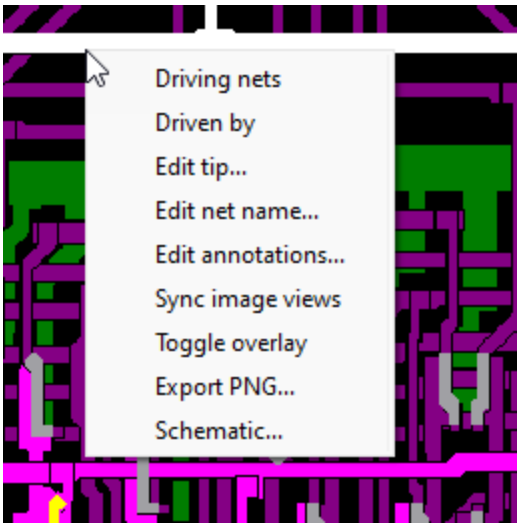
A few additional layer presets are available through the init.js script:
SHIFT + 1, +2, and +3 select certain layer combinations that may be more useful. You can edit init.js and modify it to suit your preference. Reload the script by dragging and dropping it into the app's Command Window.

The four toggle buttons (marked with 2) are:
[X] – Show or hide all active nets (nets whose current state during the simulation is logic "1"). The equivalent keyboard shortcut is "X".
[A] – Show or hide annotations. The keyboard shortcut is SPACE.
[T] – Highlight (in yellow) all active transistors. The keyboard shortcut is "T". An additional keyboard shortcut, "." (period), toggles to show you all the transistors and not just the active ones.
[L] – Show or hide detected and custom latches. The keyboard shortcut is "L".

"Find" lets you search for different features: nets by the number (for example, "398"), nets by the name ("m1"), transistors ("t2232"), and buses ("AB"). Its autocomplete helps you select a valid name.

The feature, if found, flashes on the screen and stays highlighted. Sometimes, you must zoom out to see where the feature is. Buses have only their first net highlighted. Press ESC once to clear the highlight; press ENTER key to flash the last highlight again.

As you move your mouse over the image, double-click on a net to select it, then right-click to open a context menu. The menu shows options based on the selected net and the context:



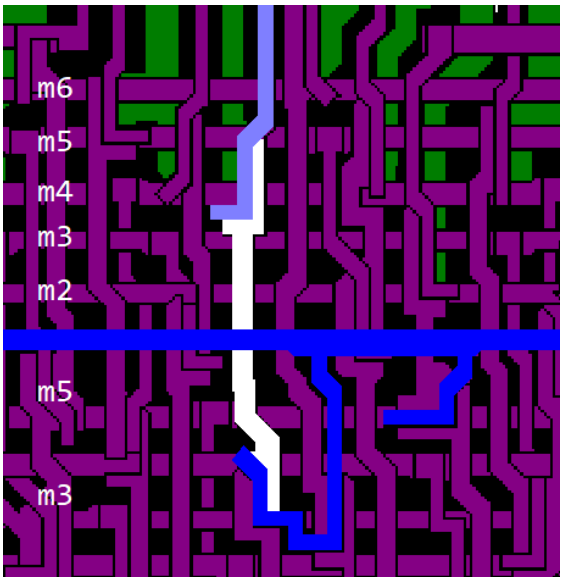
The “Sync image views” option syncs all additional views to this image location and zoom factor.
The sections below describe other options.

Driving/Driven

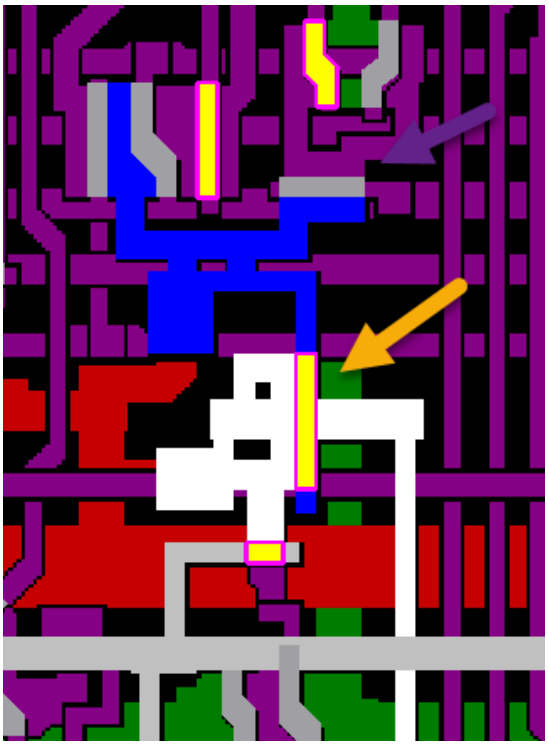
“Driving nets” and “Driven by” are two search operations on the netlist that show which nets are gated by the selected signal and which nets contribute to the selected signal. These options perform a shallow search in that they report only the first nets adjacent to the transistors and disregard serially laid-out gates such as NAND. These options are great for quickly tracing the signals.

As they are identified, the relevant nets are marked with an increasingly lighter blue hue so they can be visually discerned as part of that group. You can use these two options to trace a control signal up and down the chain of transistors and to find what causes it to change the state.

Example: net 3105 shows “Driven by” nets in blue, while the primary net (3105) is highlighted in white for a better contrast.



The picture below shows active transistors in yellow and inactive in gray. To show the transistors, press the “T” key. Sometimes, it is helpful to see all transistors lit; press “.” (period) key for that. In the image, white, having the best contrast, shows the primary selected net, blue are dependent nets and gray below is the “clk” line. The clock net is always colored gray.



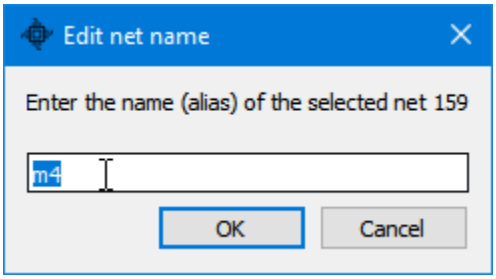
Hint: Un-select a net by double-clicking on an empty (black) space between the nets or hit the ESC key to progressively deselect them.

Edit Tips

Tips are another way to help annotate nets. Since the net names are short and abbreviated, tips provide a mechanism to expand on the meaning of a signal. Tips act as “tooltips”: as you hold your mouse over a net with a tip assigned to it, the tip shows as a mouse tooltip. Several nets have tips predefined in the default Z80 resource file. Some of the most useful ones are the PLA signal wires, for which the tips show the opcode group that a PLA wire decodes.

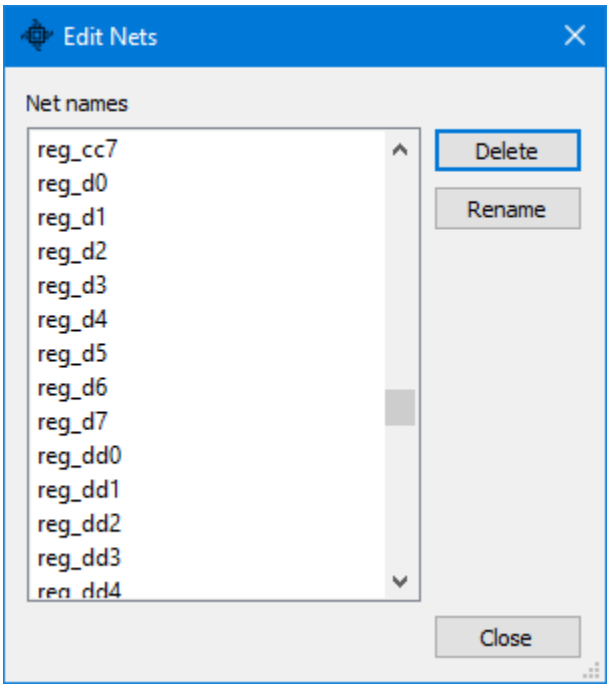
Net Names

Most of Z80’s nets are unnamed. They can be identified only by their net number. This application requires naming the nets being tracked (watched) in the simulation history and the waveform graph. Use this context menu to name and rename a selected net quickly. You can also delete the net name by clearing the input field and clicking OK.



Changes to net names are immediately visible in any waveform view that uses them.
Naming a net is a frequent operation as you figure out what it does (before adding it to the waveform view and rerunning the simulation). For example, you can give a net a temporary name (like “n287” for the net number 287), and then you can add it to the waveform view. Later, you can delete the net name if you don’t need it any longer or rename it to something more descriptive.

Another way to manage net names is to open the Edit Nets dialog with the application shortcut F2.



You can delete or rename one or more net names.

Adaptive Annotations

Custom (or user) annotations are text descriptions positioned over an image to mark a feature or show some part of interest. It is “adaptive” since, as you zoom in, larger annotations disappear to reveal smaller ones (otherwise, large text would be in the way when you zoom in.) The point at which an annotation appears and disappears depends on the size of the text and the zoom level.

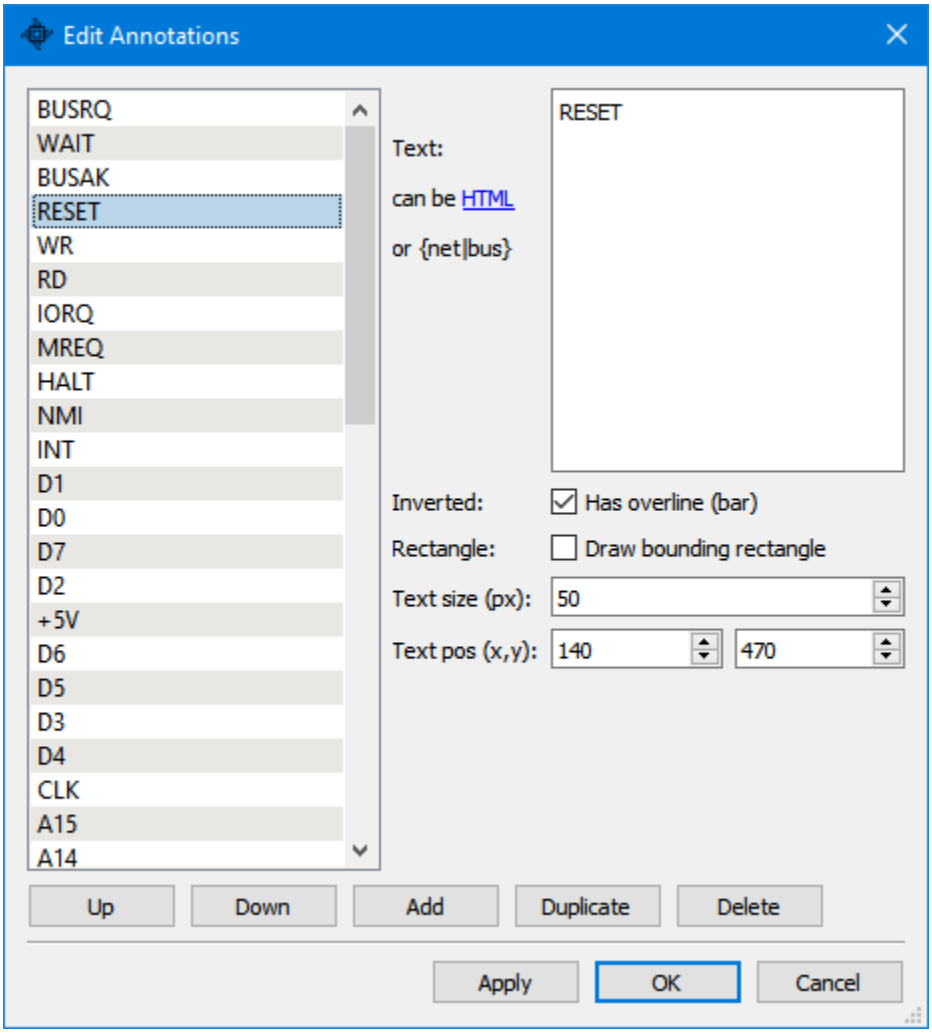
You can add and edit annotations in several ways. The simplest and most intuitive way is to right-click and drag the mouse to select the area on the image where you want to place your annotation. The size of this area also roughly defines the initial annotation text size, which you can adjust once the Edit Annotation dialog appears.

Alternatively, the application shortcut key F5 opens the Edit Annotations dialog.

If you hold down a Shift key while selecting, the selection rectangle snaps to a grid. This is useful when you are creating adjacent annotations which need to be aligned.

Using the mouse, you can select several existing annotations, and after you choose the option “Edit Annotations…”, the dialog opens with all the annotations within your selection box, all selected. Annotations can also be positioned outside the image area. Those always show and do not fade out as you zoom.

After you add an annotation, you can edit and fine-tune it if desired. Most of the time, you will re-adjust the text sizes and positions. As you adjust these parameters, click the “Apply” button to preview the changes.



The annotation text itself can include HTML-style tags (or Markdown formats). This reference website shows supported tags: https://doc.qt.io/qt-6/richtext-html-subset.html. Click on the hyperlink “can be **HTML**” within the dialog to open that reference web page in your browser.

The option “Inverted” or “overline” is helpful to tag pins with inverted input: it places a bar on top of the name.

The option “Rectangle” draws a bounding rectangle using the geometry of your original mouse selection area. This bounding rectangle cannot be changed without recreating the annotation.

By selecting multiple annotations from the list on the left, you can modify them all at once; for example, you can set the text size or one of the positions of several annotations to the same value.

You can have more than one set of annotations. While the default set loads on the application startup (“annotations.json”), you can drag and drop a file containing another set of annotations onto the Image View. Any edits will be saved back to that file. For example, “annot_internals.json” is an example of another annotation file: it contains markups of features that are less abstract than the default annotations file.

The annotation text also supports **macros**, which are simple substitutions of named nets and buses for their value. Anywhere in the annotation text, you can refer to a net or a bus, enclose its name with a set of curly brackets (for example: “{DBUS}”), and the annotation shows its current value.

A resource file, “annot_functional.json,” is an example of such a ‘functional’ annotation showing the runtime values of all major buses and latches.

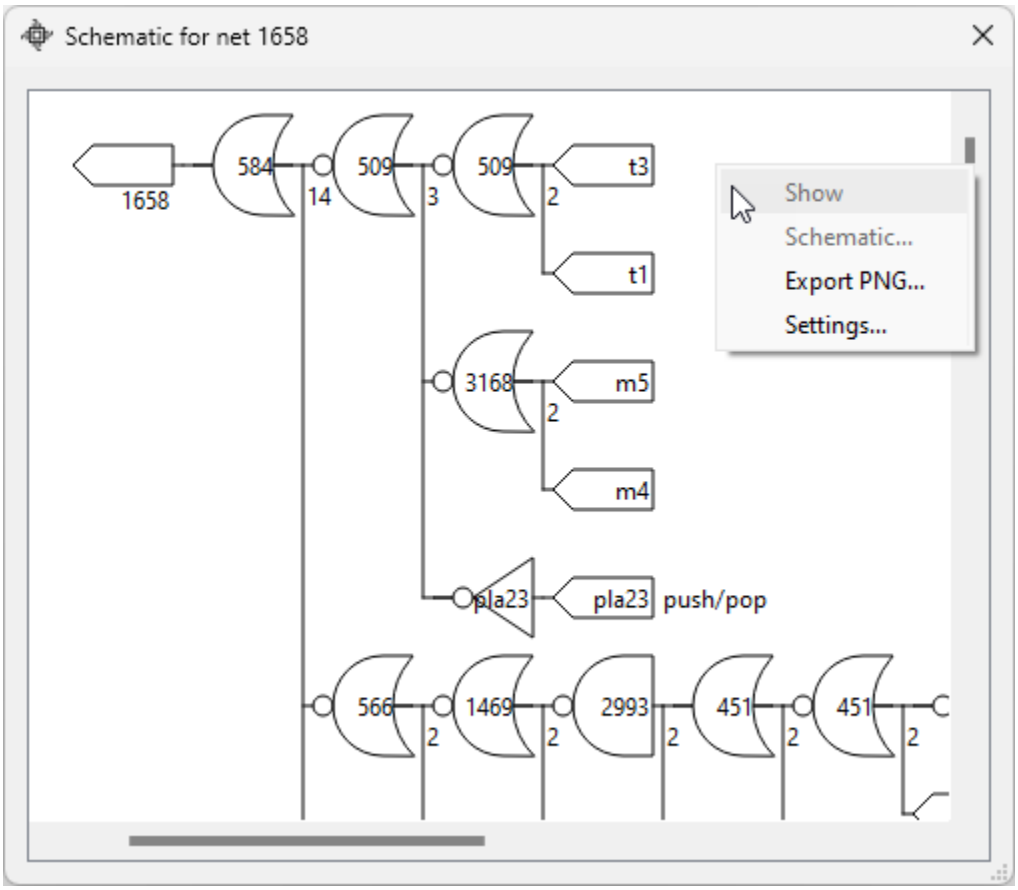
Since all internal data buses in Z80 carry inverted values, an option was added to invert the value of a net/bus when you add tilde ~ in front of a name, as in “{~VBUS}”. The value displayed is inverted, and a symbol ~ is shown to make that unambiguous. This output format is consistent with the Waveform View’s “Ones’ Complement” format.

Schematic View

This option shows the net's schematic view. At this time, it is experimental and a work in progress. The selected net is traced back to all the nets that contribute to its state. This traversal ends with certain terminating nets, which are chosen as reasonably good endpoints:

- Power, ground, and clock networks
- PLA signals
- Internal buses (ab, db, ubus, vbus)
- A few predefined nets like “int_reset” (internal reset signal)
- Detected and custom-defined latches

In addition, the traversal ends when a contributing net has already been processed to prevent possibly infinite loops. The leaf nodes display their tip text (if defined).
(1.07) The list of terminating net names is stored in file "resource/schem.ini". You can modify it by editing this file, or you can use the Schematic View's Edit dialog to manage it.

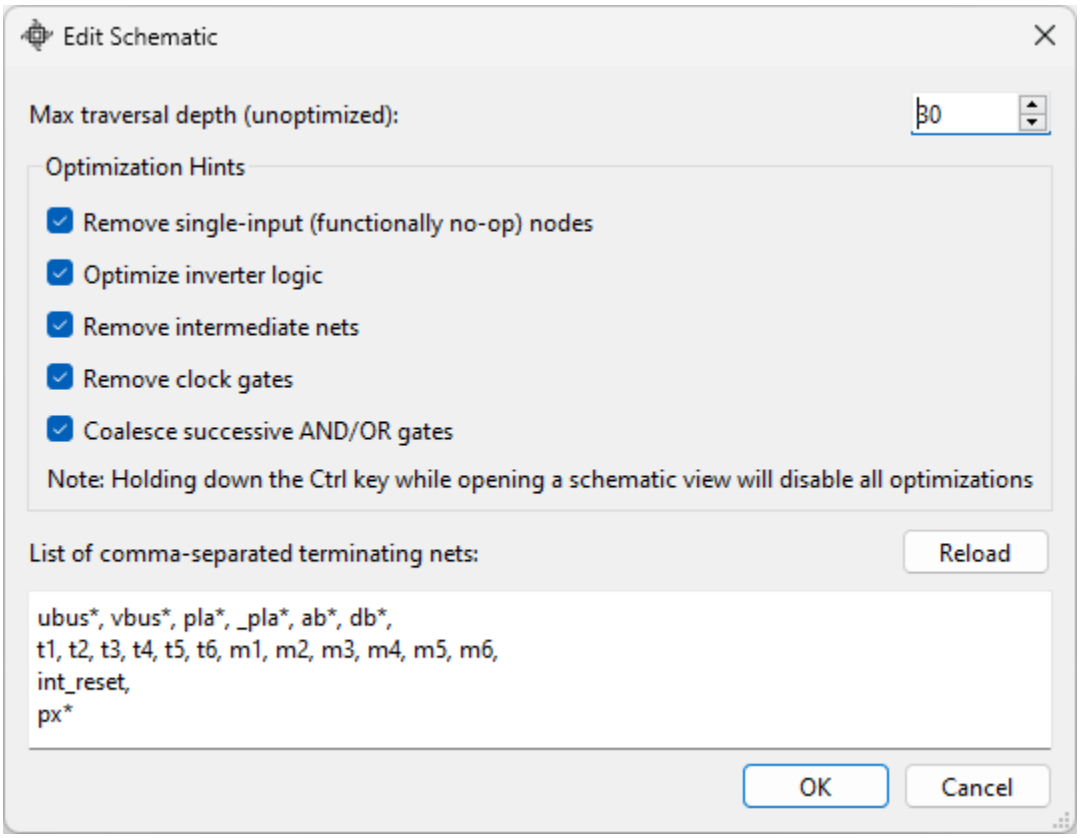


Use the mouse to pan and zoom the view. When you double-click on a logic gate, the Image view shows its feature location on the die (you may want to zoom out the Image view to see it). The corresponding context menu option is “Show”.

The context menu “Schematic…” creates a new schematic view, starting at a selected gate.

The generated logic network tree is compacted: redundant inverters are coalesced with downstream gates if possible (for example, inverter + NOR gate are collapsed into a single OR gate). This results in a somewhat smaller network. If that is not desirable, you can skip the compacting step and generate the original network if you hold down the Ctrl key while selecting the “Schematic…” menu item. In that case, the title of its window show “(not optimized”) to confirm your choice.

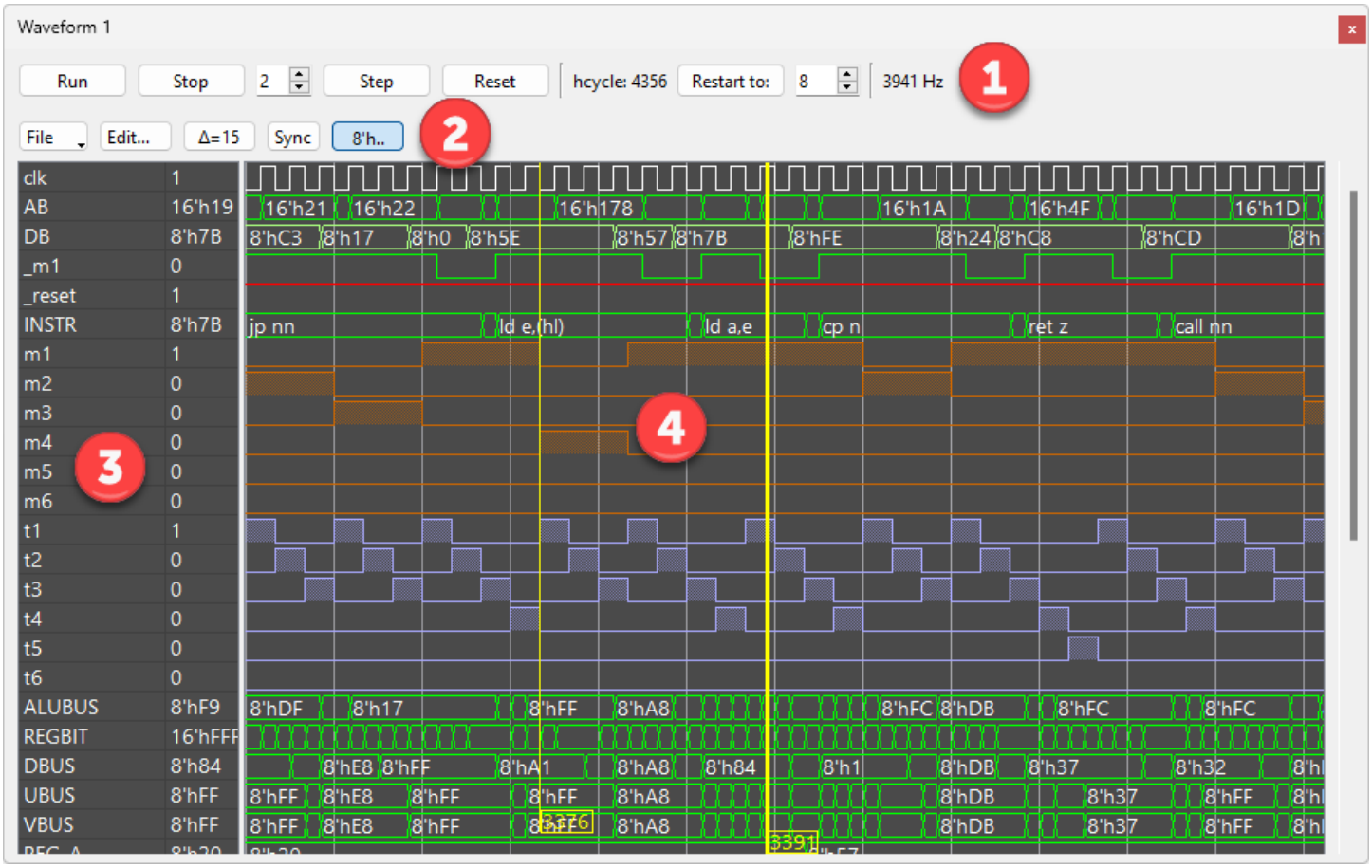
The *Schematic Edit* dialog (1.07) lets you fine-tune the schematic reverse-engineering engine.



Note: The logic parser code can detect certain features, but it is generic - and not foolproof. Trying to automatically reverse-assemble and create a schematic diagram from a chip that has been heavily hand-optimized is a difficult problem. Use this view only as a reference while still “reading” the traces yourself and using the “Driven by” option.

Waveform View

When you run a simulation, selected signal values are captured over time. A Waveform view provides a view into the history of those signals and buses even as the simulation is running:

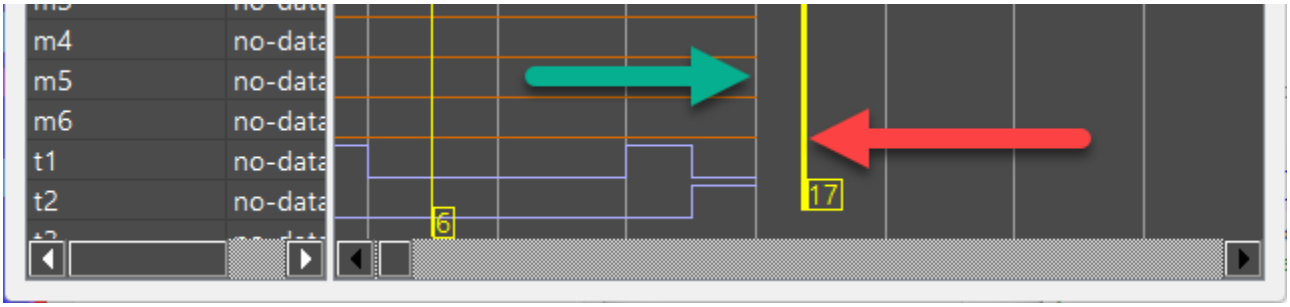


Different sections of a waveform window are:

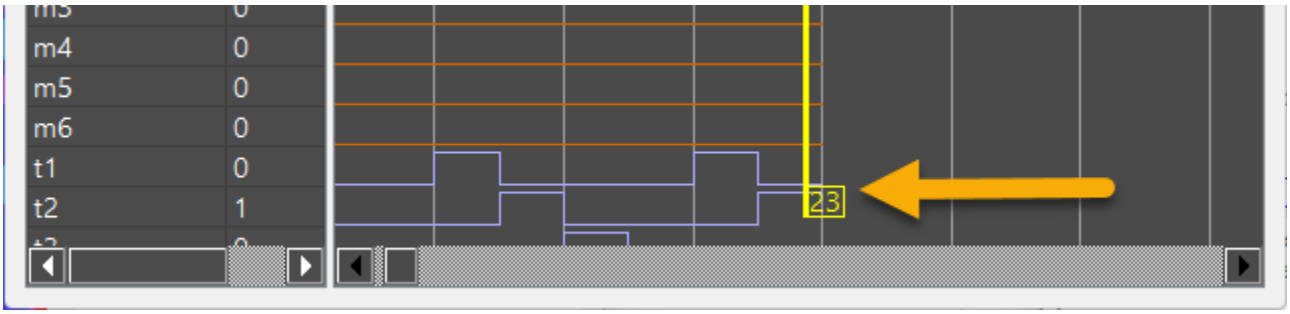
1. The simulation toolbox, identical to the main window toolbox, use it to manage simulation.
2. The file menu lets you save and load custom views (the list of nets and buses you are observing, along with their display formats). You can create several different views and load them as you need to. You can also export and save the image as a PNG file.
   Edit button opens the Edit dialog described below.
   Delta button shows the distance between the two cursors, measured in half-clock values. When that button is depressed, the two cursors link together, and the sliding one moves the other along.
   (1.07) Sync button, when depressed, will cause all Waveform windows' cursors to sync. Note that each waveform window needs to have its own Sync button also depressed to accept the sync from another window. Otherwise, sync will not affect it.
   The button marked with “8h’..” toggles bus value decorations, the bus width (in Verilog format), on and off.
3. This column shows the list of nets and buses that you are watching.

4. The central pane shows the signal history of the nets and buses. There are 2 cursors available, which you can position at any point to read out the net values.

You can place one of the cursors ahead of the signals, and as you step the execution, when the signals reach the cursor, the pane will auto-scroll to the left, leaving the signals in sight. Easier to show than describe, here you can see the leading edge of the execution (green arrow) and the active cursor (red arrow):



After we have made a few steps, the leading edge gets "pinned" to that cursor and stays visible instead of disappearing through the right edge of the window:



For that to work, the selected cursor needs to be active (highlighted), and you need to step by 1 half-steps or by 2-half-steps only.

Use the mouse or keyboard to work with the waveform window:

- The scroll wheel zooms in and out in the timeline (horizontally) (**keys**: Page Up / Page Down)
- The scroll wheel with the Ctrl key pressed enlarges the view vertically (**keys**: arrows up, down)
  (The focus needs to be on the list of nets, not on the waveform, for this to work)
- Pan left and right by dragging the pane to the extent of the available data (**keys**: arrows left, right)
- Double-click to position one of the cursors.
  You can also click on the bottom, where the cursor "flags" are, to bring in a cursor.
- Hold the cursors and move them.

You can have up to 4 separate waveform windows. Each window remembers its list of nets when you close the application. Open additional windows by pressing the Ctrl + W keyboard shortcut or via the "Window" menu.

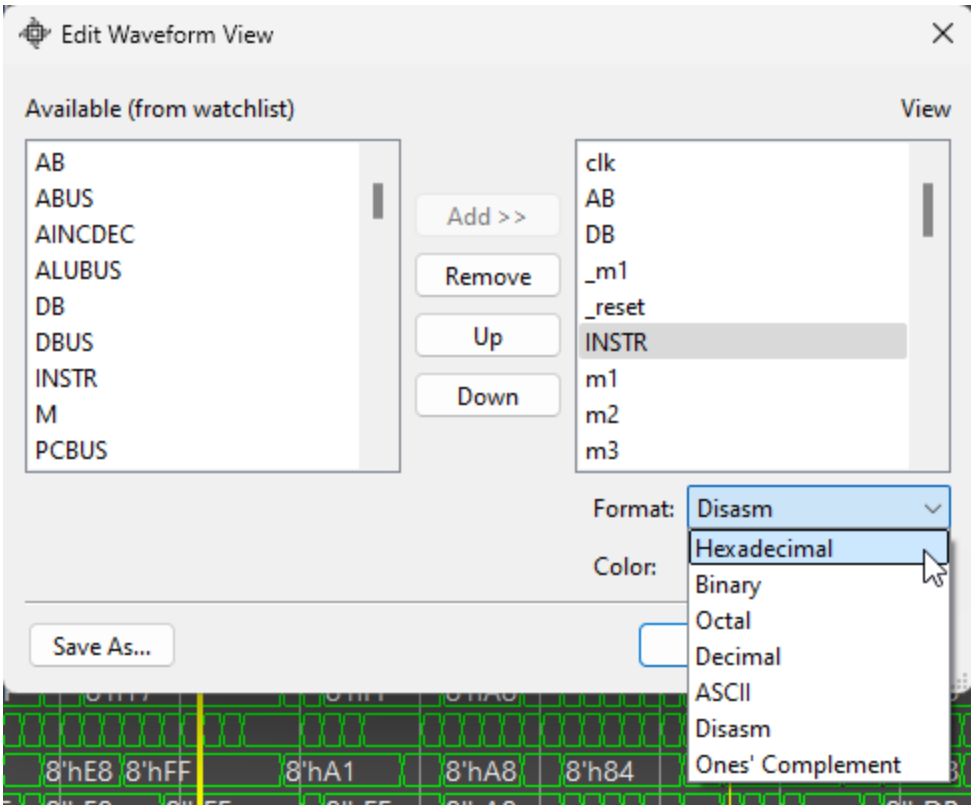
There are a few things to keep in mind:

- For any nets and buses to be available, they first need to be added to the watchlist
- That also means they need to be named (see "Net Names")
- The signal history is a rolling window containing 1000 half-cycles of data (sample points)

The Edit dialog provides a way to select which nets and buses you want to graph and its format and color. The application's convention is that the net names are lowercase, and bus names are uppercase.

(1.07) You can save current set of waveform items and Load or Merge them (from the "File" menu).

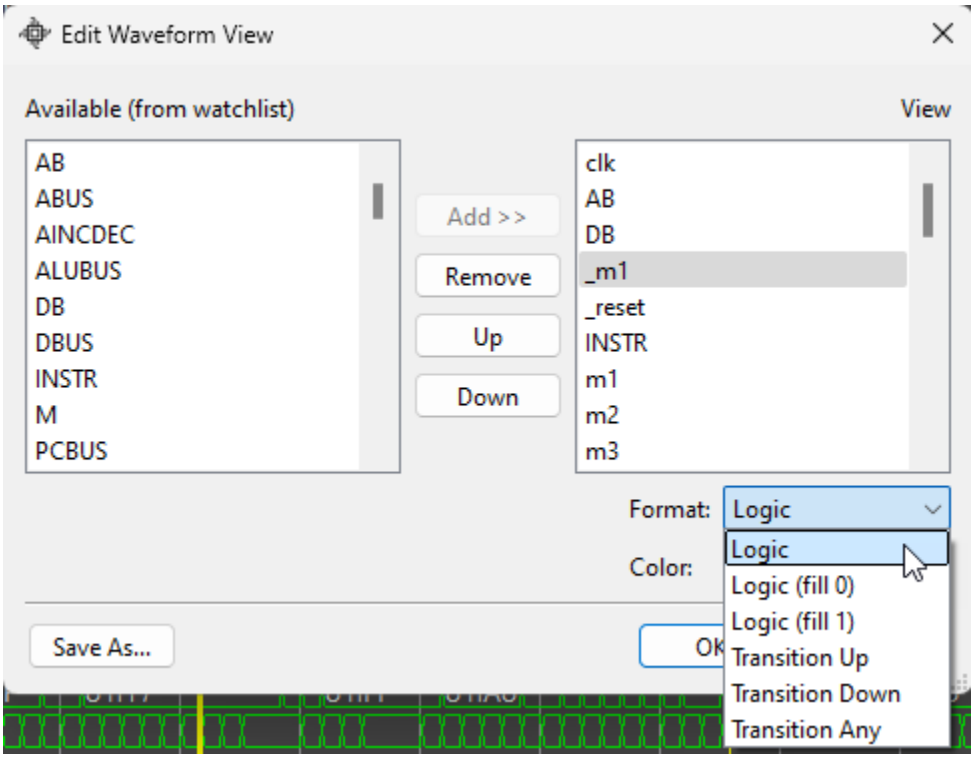
The runtime values of the buses can be shown in hexadecimal, binary, octal, decimal or ASCII, 1's Complement, and a simple Z80 dis-assembly format (which is used for instruction register):



The One's Complement format is useful when displaying internal data buses (VBUS, UBUS,…) since the bits on those buses are inverted.

Nets, only having a logical "0" or "1" value, can also have their transitions tagged:

- (1.07) The signal line is "filled" when the net is at logical "0" or "1"
- An arrow is drawn on a transition
- A simple line logic (default)



The list of Waveform view keyboard assignments:

Left/Right	Pan left/right
Up/Down	Enlarge and shrink the view vertically.
PgUp/PgDown	Zoom (also use the mouse wheel to zoom)

Waveform view mouse actions:

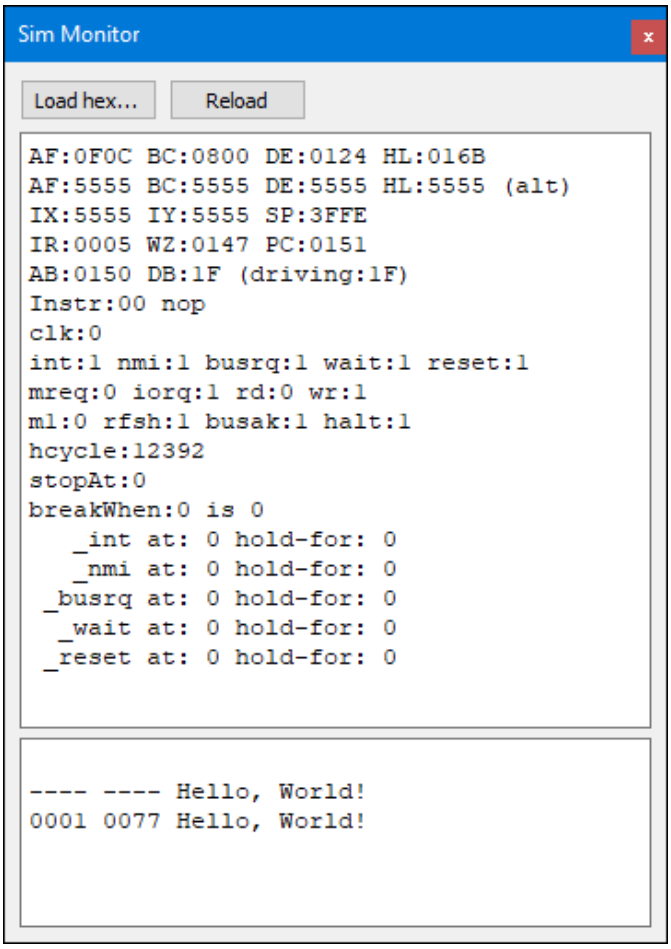
LB hold	Pan left-right; move cursors
Wheel	Zoom in and out into the history of data
Wheel + Ctrl	Enlarge and shrink the view vertically.

Sim Monitor

This window shows the simulation state and provides a terminal-like output for the executing Z80 programs to write to.

While the information listed on the top should be obvious (current values of Z80 registers and external chip pins), the values on the bottom, “stopAt” and “breakWhen” directly correspond to the values set by the scripting commands “mon.stopAt()” and “mon.breakWhen()”. Those commands, when used, stop the simulation at a specified cycle number and/or when a particular net assumes the specified value. The value of zero means the trigger has not been set or it has been cleared.

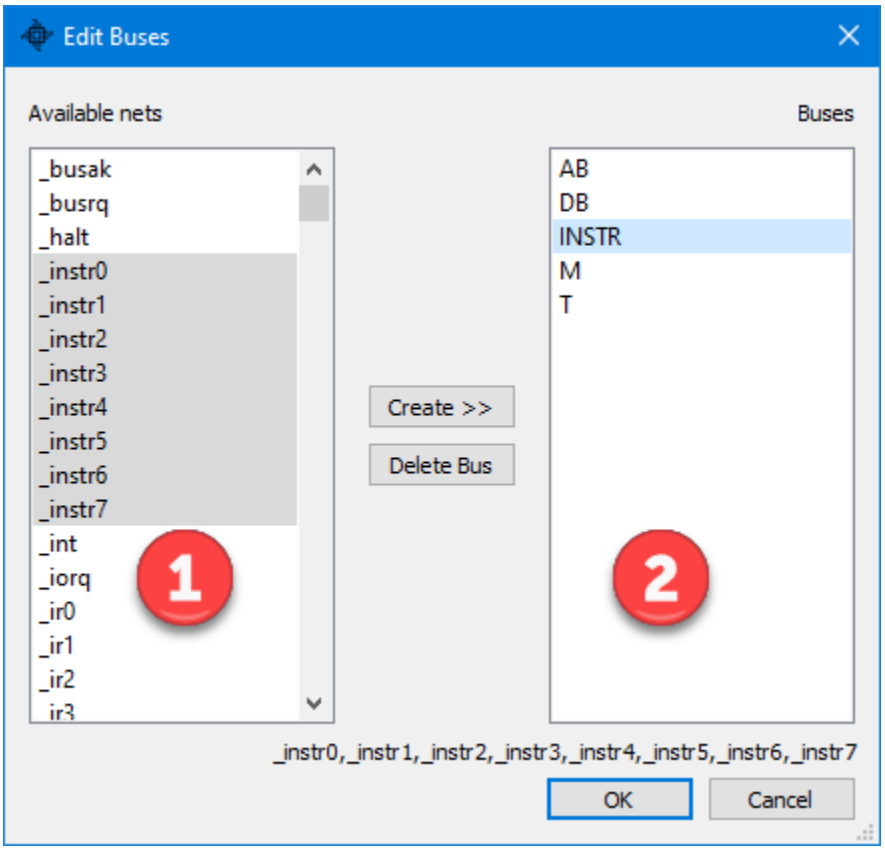
The lines listing the pin values (“_int”, …) correspond to the monitor’s memory-mapped control area: Z80 programs executing inside the simulator can cause action on these input pins. See “Simulation Environment” chapter for more details.



The bottom portion displays ASCII characters the executing Z80 program sent over its IO-mapped port 0x0800. It ignores LF (ASCII code 10) in CR/LF sequence. Writing the value of 4 (ASCII EOD, “End-of-Transmission”) to that port also causes the simulation to stop.

Edit Buses

You can combine signals into buses. Buses are two or more nets that share some logical function. The application has a few predefined buses. Bus names can be edited, or new buses can be created via the Edit Buses dialog. Press F3 at any time (or select Buses from the application's Edit menu) to open this dialog:

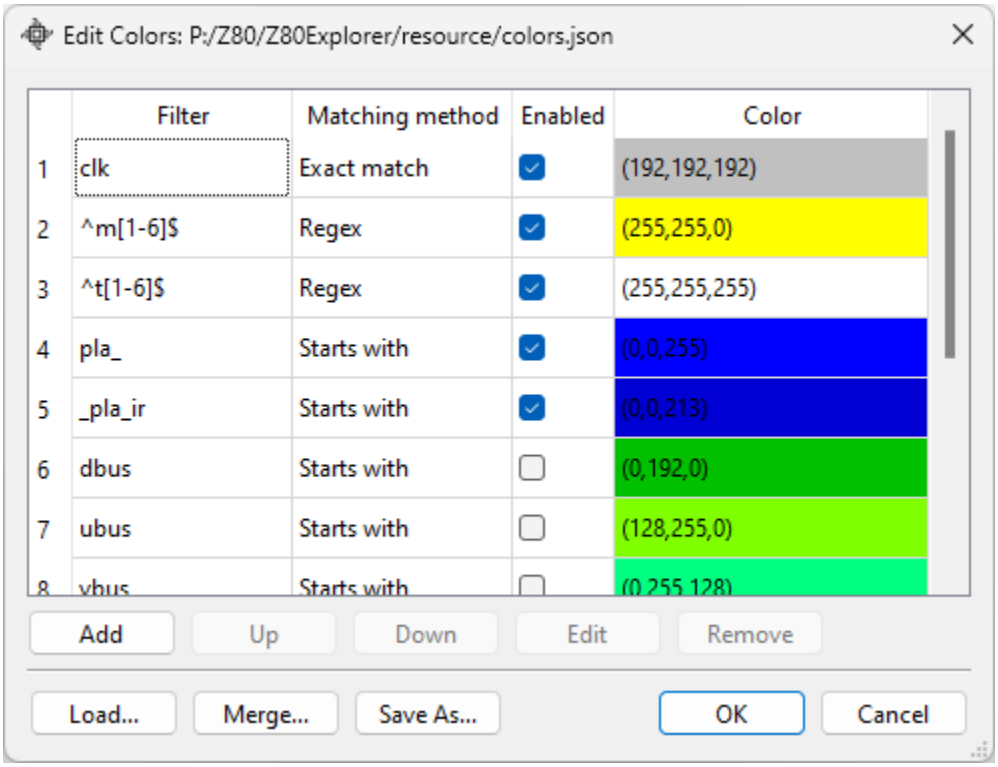


On the left is a list of (named) nets from which you can make up a bus. Press and hold the CTRL key to select each successive net. At most 16 nets can be bound into a bus. After selecting two or more, click "Create >>" to create a bus. The selected signals combine into a bus in the order you selected them.
By the application convention, all buses are written in uppercase, while the other signals (nets) are lowercase. Currently, you cannot rename a bus; the only way to effectively do that is to delete it and recreate it using a new name.

As you start defining a bus, you will see the total number of nets selected up to this point and the order of nets. This will help you create a bus faster.

Edit Colors

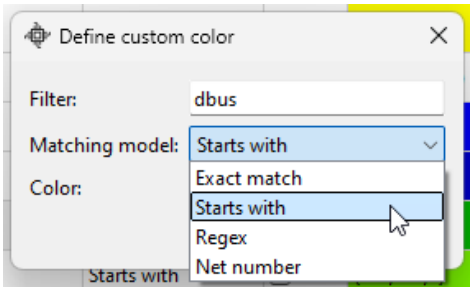
Define custom colors for the selected nets shown in the image view layer number “2”. You can select multiple color entries, move them up or down in the list, or remove them. The order on the list is essential because a net can match multiple color entries, but only the first match will be considered.



(1.07) While the application will restore last color setup (from the file "resource/colors.json"), you can save and load your own set, or merge the current set with another one. The currently active color file is displayed on the dialog title bar.

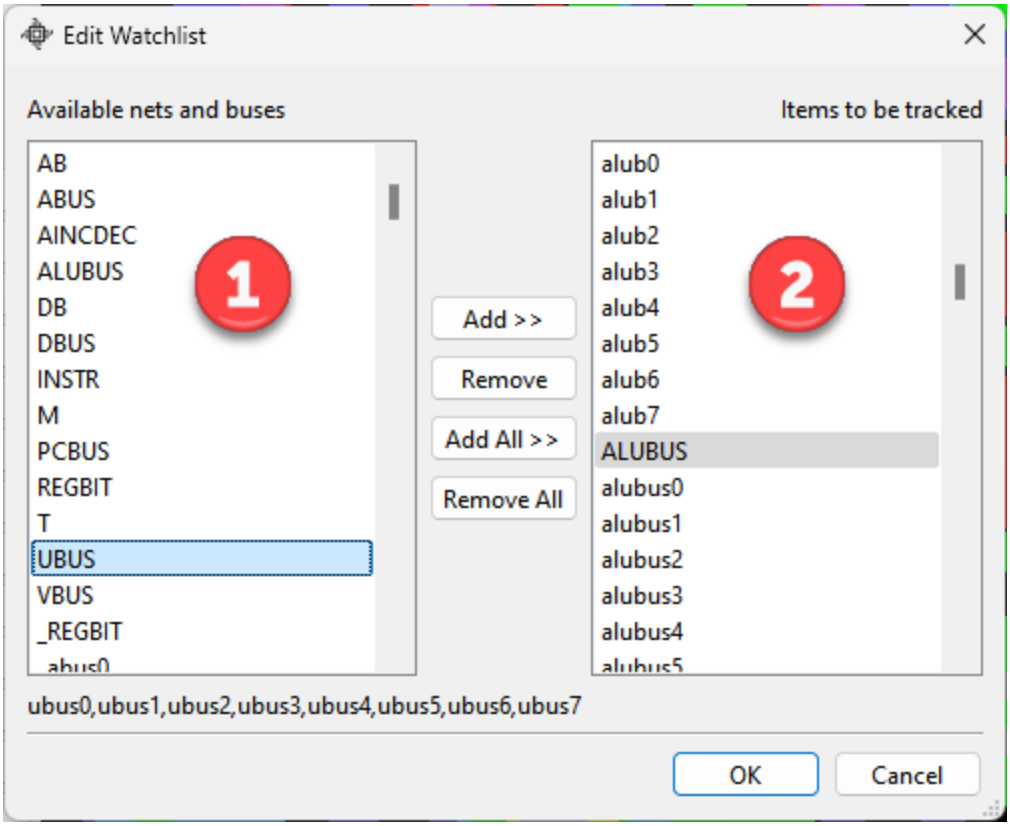
The buses and nets to be colored are selected using a filter and the corresponding matching method, which can be one of the following:

1. *Exact match*: the name in the “Filter” box has to match exactly for the color to be applied
2. *Starts with*: the net name needs to start with the specified word
3. *Regex*: use a standard regular expression to pick which nets to color
4. *Net number*: directly specify the net number to which to apply the color



Edit Watchlist

This important dialog lets you select which nets will be available to a waveform view to function correctly. Although the Z80 processor has almost 8000 nets, it is practical to keep track of only a small subset. For a net to be watched (or tracked), it must first be named (see “Net Names” section). After you name a net, it is automatically added to the watchlist. This dialog lets you fine-tune which items to track and which to leave out for the simulation performance reasons.

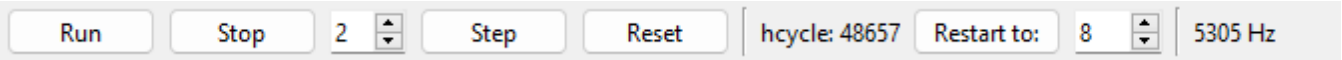


All named (and available) nets that are available to be tracked are listed on the left, and on the right side are the currently tracked ones. If you include a bus, the program also adds all the nets that make up that bus. You can select multiple entries on each side to speed up the process.

The “Add All” nets (and “Remove All” nets) buttons speed up the transition between the times when you want to run a simulation at full speed and examine the history of a run in more detail.

Running a Simulation

The simulation toolbar lets you control the simulation:



The “Run” button starts the simulation.
The “Stop” button stops (or pauses) the simulation.
The shortest simulation time is half a cycle. The default “Step” is to advance it by two half clocks or a single full clock cycle. That step time can be adjusted by changing the value in the spinbox in front of the “Step” button.
The “Reset” button clears the history and restarts the CPU. It simulates a valid power-on sequence, running for exactly 8 half-cycles (4 full cycles), after which the RESET pin goes high. You can see that process in a waveform window.
The value on the far right shows the approximate frequency of the simulation when running uninhibited.

When the application starts, it loads “hello_world.hex” test file with the Z80 code that prints “Hello, World”. The program is loaded into a simulated address space and is ready to execute.
This behavior is scripted in the “init.js” file, and you can edit it to change the startup behavior.

Several keyboard shortcuts related to running a simulation are defined in “init.js”:

Image view keyboard assignments	
F7	Step 1 half-cycle
F8	Step 2 half-cycles (one full cycle)
CTRL + F7	“Step” 1 half-cycle back (by reset and rerun)
CTRL + F8	“Step” 2 half-cycles back (by reset and rerun)

See the “init.js” file for details on this implementation. Mind that, however, the “Step back” is not a true state reversal or undoing, but the CPU is reset and then re-run up to the required cycle.

The Simulation Environment

This application runs the Z80 machine code in a simple simulated environment. The control monitor program, or “Sim Monitor,” contains a 64K RAM buffer that maps into the simulated address space. Programs (given in the Intel HEX file format) are loaded into that RAM and executed. The address to which the files are loaded is specified in a HEX file (see Intel HEX - Wikipedia). For consistency and repeatability, RAM is cleared before loading or reloading HEX data; unsuccessful loads result in the RAM containing all zeroes (equivalent to NOP instructions).

The “Load hex…” option loads a program in the HEX file format. You can also drag and drop a file onto the Sim Monitor panel to load it.

The “Reload” option reloads the last recently loaded file. This option provides a useful shortcut when you make frequent changes to a test file and want to reload it quickly.

After loading or reloading a program, you need to reset the CPU by clicking on either the “Reset” button or on the “Restart to:” button.

The monitor provides the following services **to the simulated code** by mapping specific memory address areas starting at 0xD000. Your Z80 assembly code can use the triggers provided by those services. The “test” subfolder contains several Z80 programs and the “zmac” assembler (for Windows) to create Intel HEX files suitable to load into the simulator.

For example, you can “arm” the NMI to happen at a particular clock and then stop the simulation; you can then observe what happens inside the CPU when you single-step it.

There are two ways in which the Z80 program can interact with the simulation monitor:

1) Via direct memory access – these offsets are defined in the “trickbox.inc” include file (residing in the resource test folder):

Memory Address	R/W	trickbox.inc	Description
0xD000	W	tb_stop	Writing any value to this address stops the simulation
0xD002	R/W	tb_cyc_stop	Half-cycle number at which to stop the simulation
0xD004	R	tb_cyc_low	Current clock half-cycle number (low 16 bits)
0xD006	R	tb_cyc_high	Current clock half-cycle number (high 16 bits)
0xD008	R/W	tb_int_at	Non-zero cycle number at which to assert INT pin
0xD00A	R/W	tb_int_pc	Non-zero PC address at which to assert INT pin
0xD00C	R/W	tb_int_hold	Number of cycles to hold INT asserted (default is 6)
0xD00E	R/W	tb_nmi_at	Non-zero cycle number at which to assert NMI pin
0xD010	R/W	tb_nmi_pc	Non-zero PC address at which to assert NMI pin
0xD012	R/W	tb_nmi_hold	Number of cycles to hold NMI asserted (default is 6)
0xD014	R/W	tb_busrq_at	Non-zero cycle number at which to assert BUSRQ pin
0xD016	R/W	tb_busrq_pc	Non-zero PC address at which to assert BUSRQ pin
0xD018	R/W	tb_busrq_hold	Number of cycles to hold BUSRQ asserted (default is 6)
0xD01A	R/W	tb_wait_at	Non-zero cycle number at which to assert WAIT pin
0xD01C	R/W	tb_wait_pc	Non-zero PC address at which to assert WAIT pin
0xD01E	R/W	tb_wait_hold	Number of cycles to hold WAIT asserted (default is 6)
0xD020	R/W	tb_reset_at	Non-zero cycle number at which to assert RESET pin
0xD022	R/W	tb_reset_pc	Non-zero PC address at which to assert RESET pin
0xD024	R/W	tb_reset_hold	Number of cycles to hold RESET asserted (default is 6)

When accessing these simulation control addresses, use only 16-bit wide loads and stores (“ld (**),hl”). Reading to or writing from *any other address* simulates regular RAM behavior.

Output pins (INT, NMI, BUSRQ, WAIT, and RESET) can be programmatically asserted (set to 0) either by specifying the clock cycle at which to assert them (write to any "tb_xxx_**at**" address) or by setting the PC address at which to trigger (write to any "tb_xxx_**pc**" address). You can use one of those two ways, but not both.

2) Via IO address space access:

INs and OUTs to an IO address space behave as if the IO space is another 64K segment of writable memory, independent of the addressable RAM.
OUT instructions will "write" to the IO space. IN instructions will read from the IO space. If an IO address has been "written to" (by using the OUT instruction), the subsequent IN instructions to the same address will return that value. By default, complete 64K of the IO space contains 0xFF.

The IO memory map is reset to all 0xFFs before a new program loads.

That said, there are two addresses that are treated differently. You need to read/write them by using short Z80 in/out instructions "in a,(n)" and "out (n),a". The high address byte is ignored when the low IO address byte is 0x80/0x81. Those two IO addresses are:

IO Address	IN/OUT	Description
`0x80`	OUT	Write a character to the terminal. If the character is ASCII 4 (EOT or End-of-Transmission), **stop the simulation.**
`0x81`	OUT	Byte to be presented on the data bus during the interrupt sequence in IM0 and IM2 modes.

Command Window and Scripting

Command Window provides an interactive interface to the JavaScript-based back end. It exposes several methods internal to the application. Although you do not need to use this interface to work with the application at the basic level, learning about this scripting allows you to do more in-depth analysis.

The command line keeps the history of your commands, which can be retrieved by pressing the cursor up or down keys. Hit the ESC key to clear the command line (or click on the "X" icon on the right side of the input line). Pressing the PgUp key shows the history buffer content in the application log window.

Functionally, you can drag and drop any JavaScript file (with the extension ".js") onto the Command Window to load it. That is equivalent to typing 'load("filename.js")' command.

This is a list of commands:

`load("file.js")`	`Loads and executes a JavaScript file (or "script.js" if no name is provided)`

run(hcycles)	Runs the simulation for a given number of half-cycles of the clock; use 0 to run (forever) until explicitly stopped
stop()	Stops the running simulation
reset()	Resets the simulation state
t(transistor number)	Shows a transistor state Example: t(8609)
n(net number or “name”)	Shows a net state by net number or net “name” Example: n(“t5”) or n(512)
eq(net)	Computes and shows the logic equation that drives a given net Example: n(“t5”) or n(512)
print(“string”)	Prints a string message
relatch()	Reloads all custom latches from “latches.ini” file
save()	Saves all changes to all custom and config files
ex(number)	Runs internal experimental code
execApp(...)	(1.07) Executes external application Example: execApp("notepad.exe","")

The following commands relate to the execution monitor and operate on the Z80 virtual machine:

Object “monitor”	Methods
mon.loadHex(“filename”)	Loads a HEX file into simulated memory
mon.patchHex("filename")	(1.07) Merges a HEX file into simulated memory. Use empty name for the last loaded file. Memory will *not* be cleared before loading.
mon.loadBin(“filename”, address)	Merges a binary file into sim memory at the given address. Memory will *not* be cleared before loading.
mon.saveBin(“filename”, address, size)	Saves the content of the simulated memory to a file, starting at the given address, and saving “size” bytes
mon.echo(ascii)	Echoes ASCII code to the monitor output terminal
mon.echo(“string”)	Echoes string to the monitor output terminal
mon.readMem(addr)	Reads a byte from the simulated memory
mon.writeMem(addr,value)	Writes a byte to the simulated memory
mon.readIO(addr)	Reads a byte from the simulated IO space
mon.writeIO(addr,value)	Writes a byte to the simulated IO space
mon.getHCycle()	Returns the current half-cycle simulation value
mon.stopAt(hcycle)	Stops the simulation at a given half-cycle number
mon.breakWhen(net,value)	Stops the simulation when a given net number becomes 0 or 1
mon.set(“name”,value)	Sets an output pin (*) to a value
mon.setAt(“name”,hcycle,hold)	Activates (sets to 0) an output pin (*) at a specified half-cycle and holds it for the number of half-cycles
mon.setPC(“name”,addr,hold)	Activates (sets to 0) an output pin (*) when PC equals the address and hold it for the number of half-cycles
mon.enabled = 1|0 true|false	(variable) Enables or disables monitor’s memory-mapped services at the address 0xD000
mon.rom = value	(variable) Designates the initial number of bytes for the read-only memory region; default 0. Example:

	monitor.rom = 8192 designates 0-8191 as a non-writable region

* Output pins: "int", "nmi", "busrq", "wait", "reset"

The following commands interact with the image window of the main application (not with any extra image views):

Object “image”	Methods
img.setLayer(“id”)	Sets the layer id (“1”...”k”)
img.addLayer(“id”)	Adds the layer id (“1”...”k”) to the one(s) already set
img.setZoom(value)	Sets the zoom value (from 0.1 to 10.0)
img.setPos(x, y)	Moves the image to coordinates x: 0 – 4700, y: 0 – 5000
img.find(“feature”)	Finds and shows the named feature Ex. img.find(“260”)
img.show(x,y,w,h)	Highlight a rectangle at the given coordinates and width, height. You can read those values in the log window after selecting an image area using the mouse.
img.state()	Prints the current image view position and zoom level to the log window as a command string suitable to copy and use later to restore the exact image view
img.annot(“filename.json”)	Loads a custom annotation file to all image views

When the application starts, it loads a JavaScript startup script called “init.js”. This file should be found in the resource folder. That script defines a few helpful functions and loads a “Hello, World” Z80 program. Run it by clicking the toolbox’s “Run” button.

This script also redefines some monitor and image class methods into more straightforward JavaScript functions. For example, you may write “state()” instead of “img.state()”. That only works for functions, not for variables. Type “help()” to get the list of implemented functions.

Ensure the JavaScript contains no long-running code since the scripting engine blocks the rest of the application until the running script functions return. That does not include the commands to start the simulation run.

Network Socket Server

(1.07) For safety, this feature is disabled in the default build. If you want to use it, change the define "SOCKET_SERVER" (in AppTypes.h) to 1 and rebuild the application.

The application opens a socket server at the port 12345 (the port number is hard-coded at the moment in file ClassController.cpp). A client can connect and issue commands remotely. An example Python script, "sock.py" implements a basic command client.

Unsorted Notes and Tidbits

The application stores its settings (windows positions, sizes, …) in the Windows registry at this path:
`HKEY_CURRENT_USER\Software\Baltazar Studios, LLC\Z80Explorer`
On Linux, it stores them in a folder under your user's home:
`~/.config/Baltazar Studios, LLC`

In this order, hitting the ESC key clears highlighted nets, "driven" nets, and the selected net.

The application supports touch and multi-touch devices. You can scroll (drag) and pinch-to-zoom on images and graphics views.

The application tries to detect latches in the netlist by a simple heuristic of finding two adjacent nets gating each other. While that alone detects many latches, many are still undetected. Hence, the resource file "latches.ini" provides a way to define additional latches as you discover them. It specifies a list of two transistor numbers for each additional latch. See the file itself for an example of how to specify a latch. You can edit it while the application is running. After changing it externally, you must reload it by typing "relatch()" in the script command window.

Everything in the simulation is measured in half-cycles: the high and low clock are two distinct states. The documentation, however, interchangeably uses the terms "half-cycle", "hcycle" and even "cycle" for brevity.

List of Resource Files

Many resource files have a convenient JSON format (a text file format). The application will load and save some of these files on exit, so if you need to edit them on your own, do it while the application is not running.

File	Use
annotations.json	List of annotations and their properties
annot_internals.json	Alternate annotation set; load by dropping onto the Image view
annot_functional.json	Another annotation file showing major functional buses and nets
colors.json	List of color definitions, filters and matching methods
init.js	Startup script file
latches.ini	List of additional latches beyond those that are auto-detected
netnames.js	List of custom added, net and bus names
tips.json	List of user net tips
watchlist.json	List of nets that are being watched / tracked
waveform-*.json	State of each of the four waveform windows

Extras: ZX Spectrum, …

Load “zx.js” file (or drop it into the Command Window) and the application starts modeling Sinclair ZX Spectrum. It opens a view into its simulated screen and it starts running the ROM code. On a fast PC, it takes about 20 minutes for the main screen to get cleared and to have the Sinclair logo appear.

The resource folder also contains “zexall.hex” which you can load and run. The tests are sorted by the time they take to execute.

A number of other tests from that folder can be assembled and run using the “zmac.exe” assembler.

(1.07) Load SNA file format: `load("sna.js")`
See the "sna.js" file for details or modifications. Issuing this command loads a SNA file and starts running it. A window opens at the point where you should be able to see how this game updates the screen: look for the lines and pixels being updated. It would be nice to be able to play it, wouldn't it?

Known Issues

JavaScript engine:
Ensure the JavaScript contains no long-running code since the scripting engine blocks the rest of the application from running.

Schematic view:
Functional issues: Logic tree parser is a work in progress; the application may decode some nodes incorrectly. Always check.

Credits

This application heavily builds on the work done by the Visual6502 team: Chris Smith, Ed Spittles, Pavel Zima et al.: http://www.visual6502.org
The Visual 6502 team also created main Z80 image layers and many initial net names.