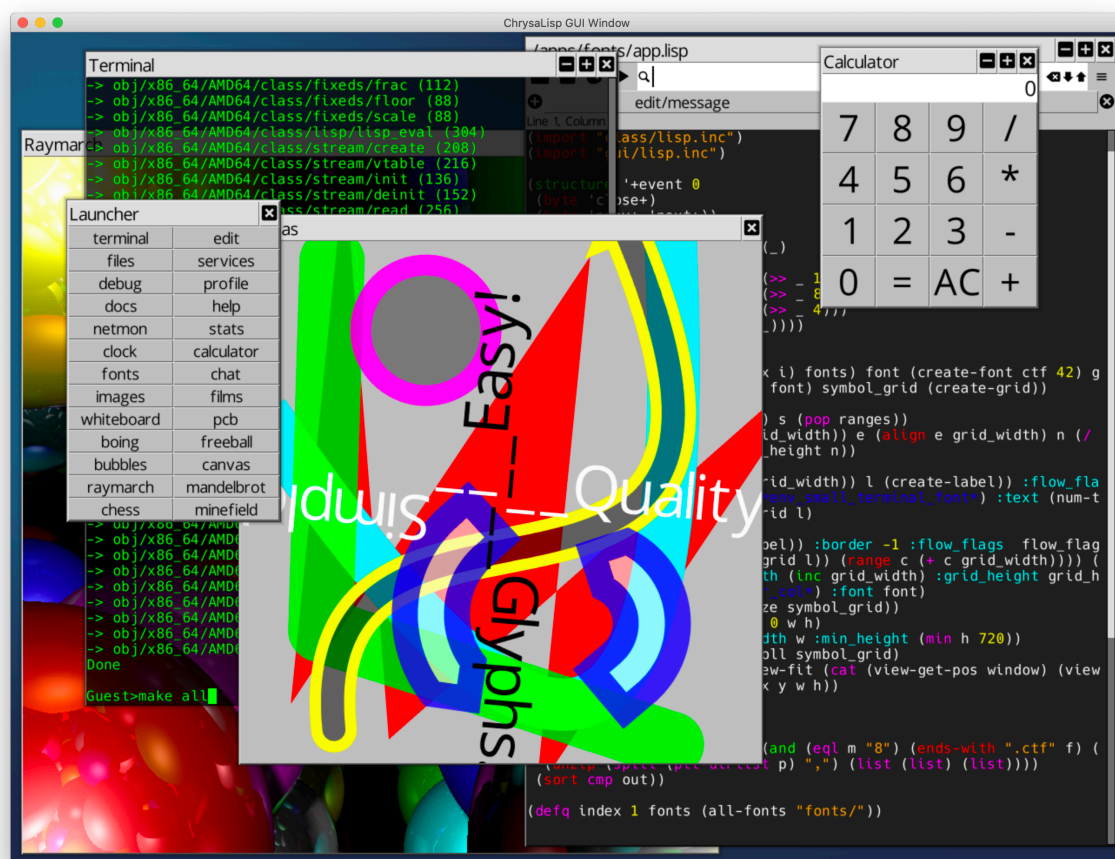
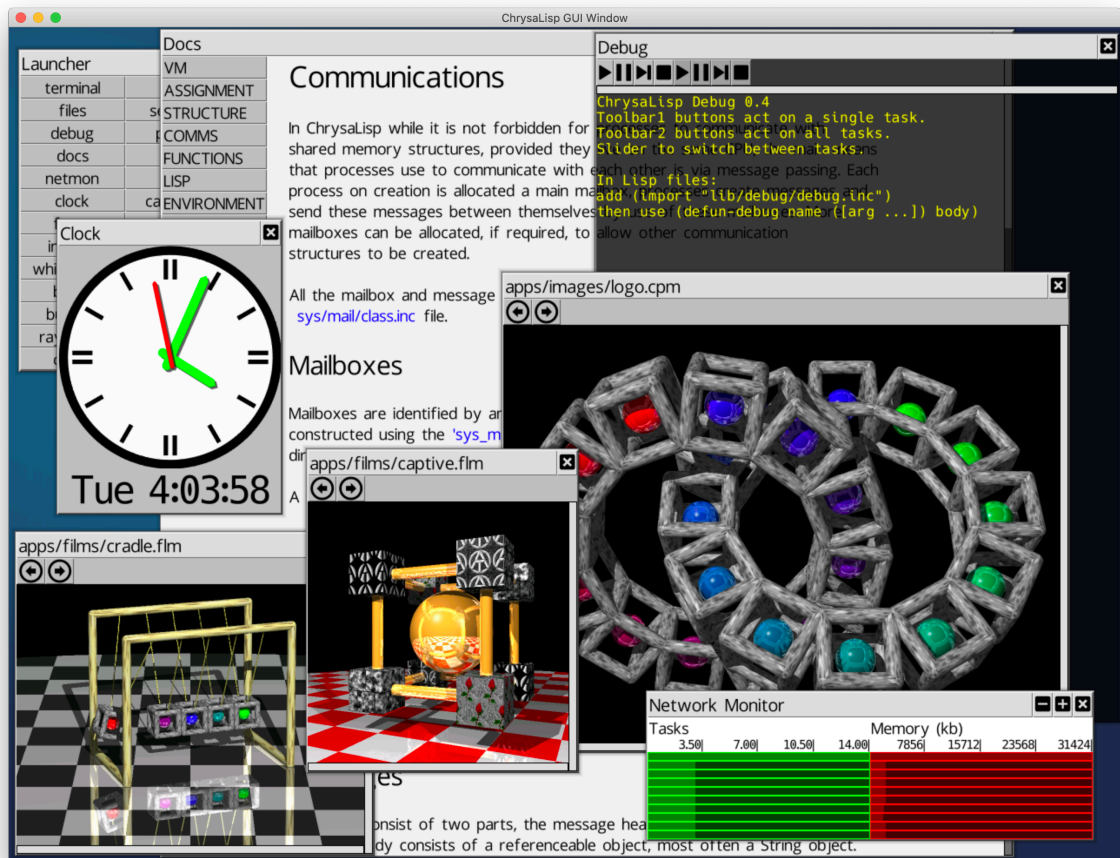
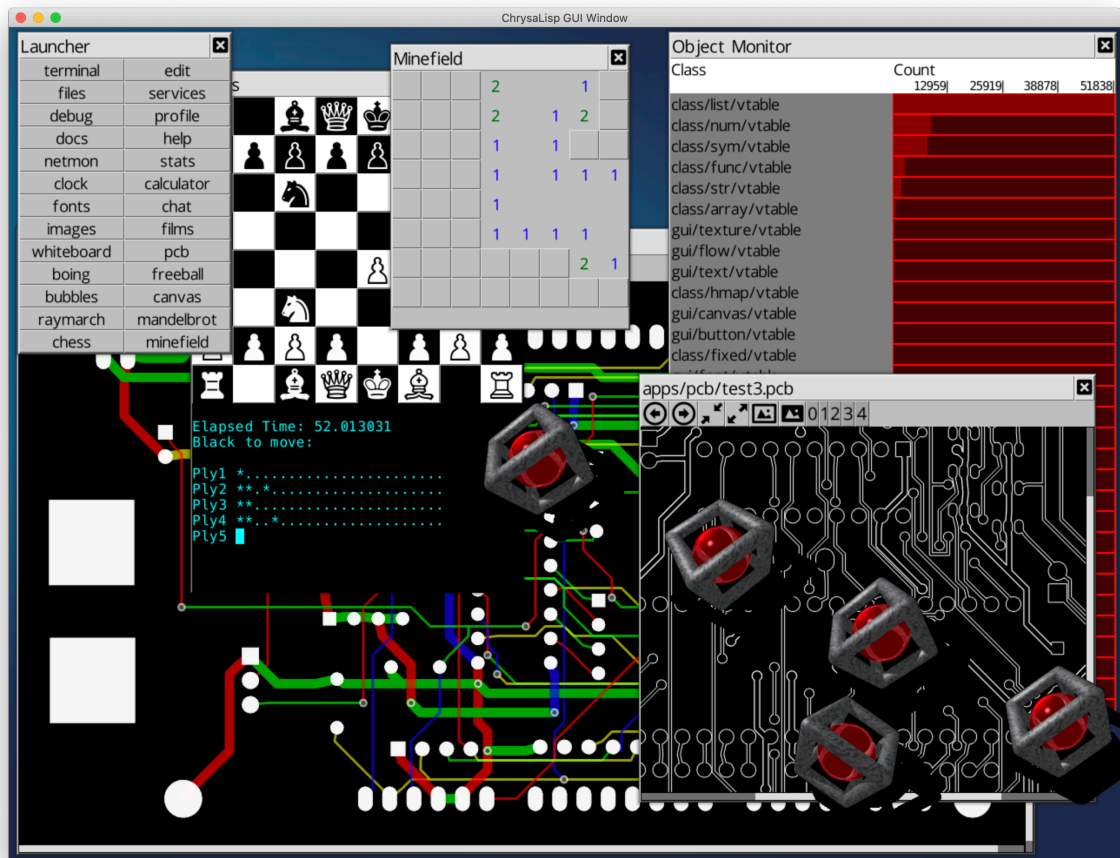


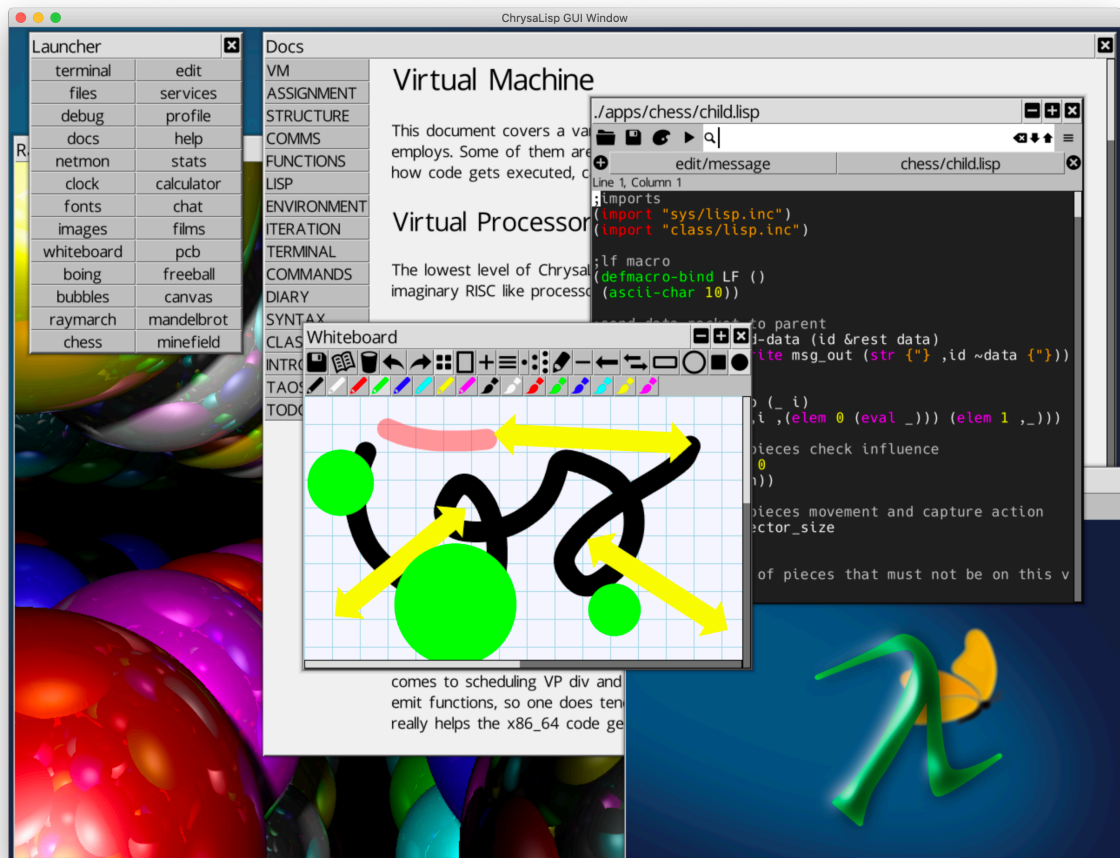
ChrysaLisp

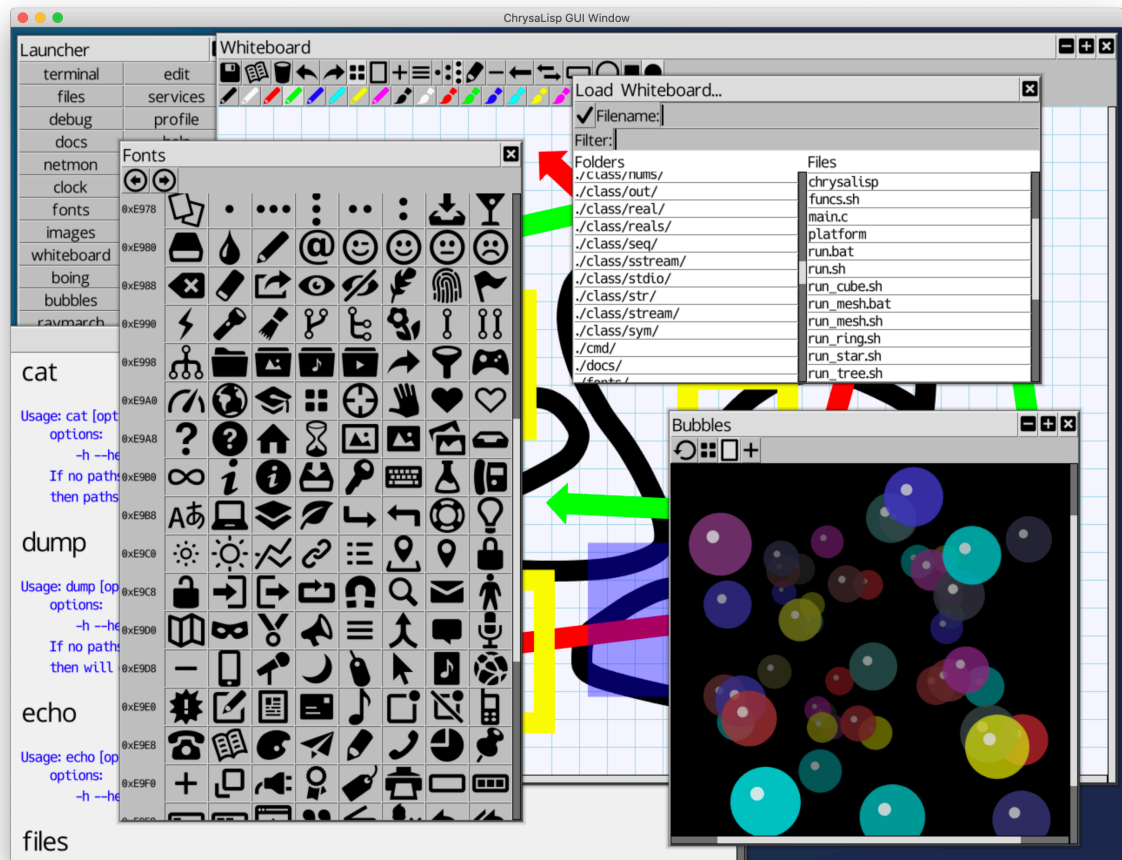
ChrysaLisp is a 64-bit, MIMD, multi-CPU, multi-threaded, multi-core, multi-user parallel operating system with features such as a GUI, terminal, OO Assembler, class libraries, C-Script compiler, Lisp interpreter, debugger, profiler, vector font engine, and more. It supports MacOS, Windows, and Linux for x64, Riscv64 and Arm64 and eventually will move to bare metal. It also allows the modeling of various network topologies and the use of ChrysaLib hub_nodes to join heterogeneous host networks. It has a virtual CPU instruction set and a powerful object and class system for the assembler and high-level languages. It has function-level dynamic binding and loading and a command terminal with a familiar interface for pipe-style command line applications. A Common Lisp-like interpreter is also provided.

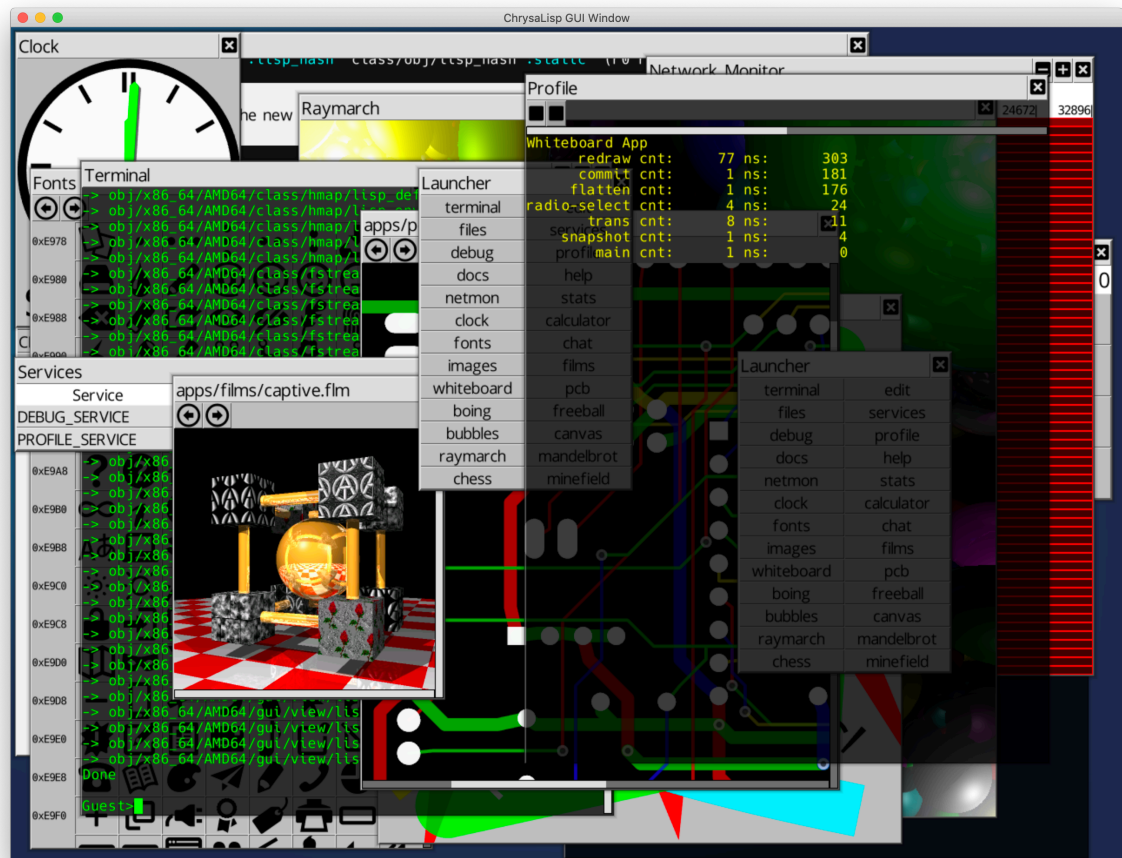












8

supports a VP64 software CPU emulator used for the install process, but this can be used, with the `-e` option, on platforms where no native CPU support currently exists, as the runtime system. It runs on a hosted environment while experimentation is being done, but eventually it will be transitioned to run on bare metal. In the future, I plan to create a VM boot image for UniKernel appliances and a WebAssembly target for use within web browsers.

ChrysaLisp allows for the simulation of various network topologies utilizing point-to-point links. Each CPU in the network is represented as a separate host process, and point-to-point links utilize shared memory to simulate CPU-to-CPU, bidirectional connections. The design intentionally does not include global bus-based networking.

The ChrysaLib project, <https://github.com/vygr/ChrysaLib>, enables the use of IP and USB3/USB2 Prolific chip “copy” cables to create heterogeneous host networks. This allows users to connect their MacBooks, Linux, Windows machines and PI4’s to create their own development LAN or WAN networks, which is pretty cool.

ChrysaLisp uses a virtual CPU instruction set to eliminate the use of x64, ARM64, RISCv64, or VP64 native instructions. Currently, it compiles directly to native code, but it has the capability to also be translated to byte code form and use runtime translation.

To avoid the need for register juggling for parameter passing, all functions define their register interface, and parameter sources and destinations are automatically mapped using a topological sort. If non-DAG mappings are detected, the user can address them with a temporary. The software also includes operators to make it easy to bind parameters to dynamic bound functions, relative addresses, auto-defined string pools, references, and local stack frame values. Output parameters that are not used can be ignored with an underscore.

ChrysaLisp has a powerful object and class system that is not limited to just the assembler but is quite as capable as a high level language. It allows for the definition of static classes or virtual classes with inline, virtual, final, static, and override methods. The GUI and Lisp are built using this class system.

It has function-level dynamic binding and loading. Functions are loaded and bound on demand as tasks are created and distributed. Currently, functions are loaded from the CPU file system where the task is located, but in the future, they will come from the server object that the task was created with and will be transported across the network as needed. Functions are shared among all tasks that use the same server object, so only one copy of a function is loaded, regardless of how many tasks use it.

The system functions are accessed through a set of static classes, which makes it easy to use and eliminates the need to remember static function locations, and also decouples the source from changes at the system level. The interface definitions for these functions can be found in the *sys/xxx.inc* files.

A command terminal with a familiar interface for pipe style command line applications is provided

with args vector, stdin, stdout, stderr etc. Classes for easy construction of pipe masters and slaves, with arbitrary nesting of command line pipes. While this isn't the best way to create parallel applications it is very useful for the composition of tools and hides all the message passing behind a familiar streams based API.

A Common Lisp like interpreter is provided. This is available from the command line, via the command `lisp`. To build the entire system type `(make)`, calculates minimum compile workload, or `(make-all)` to do everything regardless, at the Lisp command prompt. This Lisp has a C-Script 'snippets' capability to allow mixing of C-Script compiled expressions within assignment and function calling code. An elementary optimize pass exists for these expressions. Both the virtual assembler and C-Script compiler are written in Lisp, look in the *lib/asm/code.inc*, *lib/asm/xxx.inc*, *lib/asm/-func.inc*, *lib/trans/x86_64.inc*, *lib/trans/arm64.inc* and *lib/asm/vp.inc* for how this is done. Some of the Lisp primitives are constructed via a boot script that each instance of a Lisp class runs on construction, see *class/lisp/root.inc* for details. The compilation and make environment, along with all the compile and make commands are created via the Lisp command line tool in *lib/asm/asm.inc*, again this auto runs for each instance of the `lisp` command run from the terminal. You can extend this with any number of additional files, just place them after the `lisp` command and they will execute after the *lib/asm/asm.inc* file and before processing of stdin.

Don't get the idea that due to being coded in interpreted Lisp the assembler and compiler will be slow. A fully cleaned system build from source, including creation of a full recursive pre-bound boot image file, takes on the order of 2 seconds on a 2014 MacBook Pro ! Dev cycle `(make)` and `(remake)` under 0.5 seconds. It ain't slow !

Network link routing tables are created on booting a link, and the process is distributed in nature, each link starts a flood fill that eventually reaches all the CPU's and along the way has marked all the routes from one CPU to another. All shortest routes are found, messages going off CPU are assigned to a link as the link becomes free and multiple links can and do route messages over parallel routes simultaneously. Large messages are broken into smaller fragments on sending and reconstructed at the destination to maximize use of available routes.

The `-run` command line option launches tasks on booting that CPU, such as the experimental GUI (a work in progress, `-run gui/gui/gui.lisp`). You can change the network launch script to run more than one GUI session if you want, try launching the GUI on more than CPU 0, look in *funcs.sh* at the `boot_cpu_gui` function ! :)

The `-l` command line option creates a link, currently up to 1000 CPU's are allowed but that's easy to adjust. The shared memory link files are created in the tmp folder */tmp*, so for example */tmp/000-001* would be the link file for the link between CPU 000 and 001.

An example network viewed with `ps` looks like this for a 4x4 mesh network:

```
1 ./main_gui -l 011-015 -l 003-015 -l 014-015 -l 012-015
2 ./main_gui -l 010-014 -l 002-014 -l 013-014 -l 014-015
3 ./main_gui -l 009-013 -l 001-013 -l 012-013 -l 013-014
4 ./main_gui -l 008-012 -l 000-012 -l 012-015 -l 012-013
5 ./main_gui -l 007-011 -l 011-015 -l 010-011 -l 008-011
6 ./main_gui -l 006-010 -l 010-014 -l 009-010 -l 010-011
7 ./main_gui -l 005-009 -l 009-013 -l 008-009 -l 009-010
8 ./main_gui -l 004-008 -l 008-012 -l 008-011 -l 008-009
9 ./main_gui -l 003-007 -l 007-011 -l 006-007 -l 004-007
10 ./main_gui -l 002-006 -l 006-010 -l 005-006 -l 006-007
11 ./main_gui -l 001-005 -l 005-009 -l 004-005 -l 005-006
12 ./main_gui -l 000-004 -l 004-008 -l 004-007 -l 004-005
13 ./main_gui -l 003-015 -l 003-007 -l 002-003 -l 000-003
14 ./main_gui -l 002-014 -l 002-006 -l 001-002 -l 002-003
15 ./main_gui -l 001-013 -l 001-005 -l 000-001 -l 001-002
16 ./main_gui -l 000-012 -l 000-004 -l 000-003 -l 000-001 -run gui/gui
```

Getting Started

Take a look at the [docs/intro.md](#) for instructions to get started on all the supported platforms.

The experimental GUI requires the **SDL2** library to be installed.

Get them via your package manager, on Linux with:

```
1 sudo apt-get install libsdl2-dev
```

Or on Mac via Homebrew.

```
1 brew install sdl2
```

Make/Run/Stop

Take a look at the [docs/intro/intro.md](#) for platform specific instructions. The following is for OSX and Linux systems. Windows has a pre-built main.exe provided, or you can configure Visual Studio to compile things yourself if you wish.

Installing

The first time you download ChrysaLisp you will only have the vp64 emulator boot image. You must create the native boot images the first time round. This is a little slower than subsequent boots and system compiles but allows us to keep the snapshot.zip file as small as possible.

If on Linux or Mac via Homebrew:

```
1 make install
```

Or on Windows

```
1 install.bat
```

Make

```
1 make
```

Run

```
1 ./run_tui.sh [-n num_cpus] [-e] [-b base_cpu]
```

Text user interface based fully connected network. Each CPU has links to every other CPU. Careful with this as you can end up with a very large number of link files and shared memory regions. CPU 0 launches a terminal to the host system.

```
1 ./run.sh [-n num_cpus] [-e] [-b base_cpu]
```

Fully connected network. Each CPU has links to every other CPU. Careful with this as you can end up with a very large number of link files and shared memory regions. CPU 0 launches a GUI.

```
1 ./run_star.sh [-n num_cpus] [-e] [-b base_cpu]
```

Star connected network. Each CPU has a link to the first CPU. CPU 0 launches a GUI.

```
1 ./run_ring.sh [-n num_cpus] [-e] [-b base_cpu]
```

Ring connected network. Each CPU has links to the next and previous CPU's. CPU 0 launches a GUI.

```
1 ./run_tree.sh [-n num_cpus] [-e] [-b base_cpu]
```

Tree connected network. Each CPU has links to its parent CPU and up to two child CPU's. CPU 0 launches a GUI.

```
1 ./run_mesh.sh [-n num_cpus on a side] [-e] [-b base_cpu]
```

Mesh connected network. Each CPU has links to 4 adjacent CPU's. This is similar to Transputer meshes. CPU 0 launches a GUI.

```
1 ./run_cube.sh [-n num_cpus on a side] [-e] [-b base_cpu]
```

Cube connected network. Each CPU has links to 6 adjacent CPU's. This is similar to TMS320C40 meshes. CPU 0 launches a GUI.

Stop

Stop with:

```
1 ./stop.sh
```

Snapshot

Snapshot with:

```
1 make snapshot
```

This will create a *snapshot.zip* file of the *obj/* directory containing only the host directory structures, the pre-compiled Windows *main_gui.exe* and *main_tui.exe* plus the VP64 *boot_image* files !

Used to create the more compact *snapshot.zip* that goes up on Github. This must come after creation of `(make-all-platforms)` *boot_image* set !

```
1 obj/vp64/VP64/sys/boot_image
2 obj/x86_64/WIN64/Windows/main_gui.exe
3 obj/x86_64/WIN64/Windows/main_tui.exe
```

Clean

Clean with:

```
1 make clean
```