SimpLC Project

Search this site

Overview

Downloads: on Sourceforge

Mission: To provide a pleasant and enriching debugging experience for beginning programmers of all assembly languages.

Status: Currently sports Qt and console interfaces, already providing LC-3 and Georgia Tech's LC2200 plus an extensible framework for more. There are plans to make it a superior alternative to SPIM.

Contact: garryb@gmail.com (Garry Boyer)

Downloads: See the SourceForge summary page and click on the green downloads bar.

History: Began as a project to provide a better LC-3 simulator experience to CS2110 students at Georgia Tech. After creating a very extensive user interface, the LC-3 simulator evolved to allow other architectures to plug in. Now it supports LC-3 and the old 8-bit LC-2200, plus a Mips simulator is in development, and interested developers can add anything else.

Feature Comparison

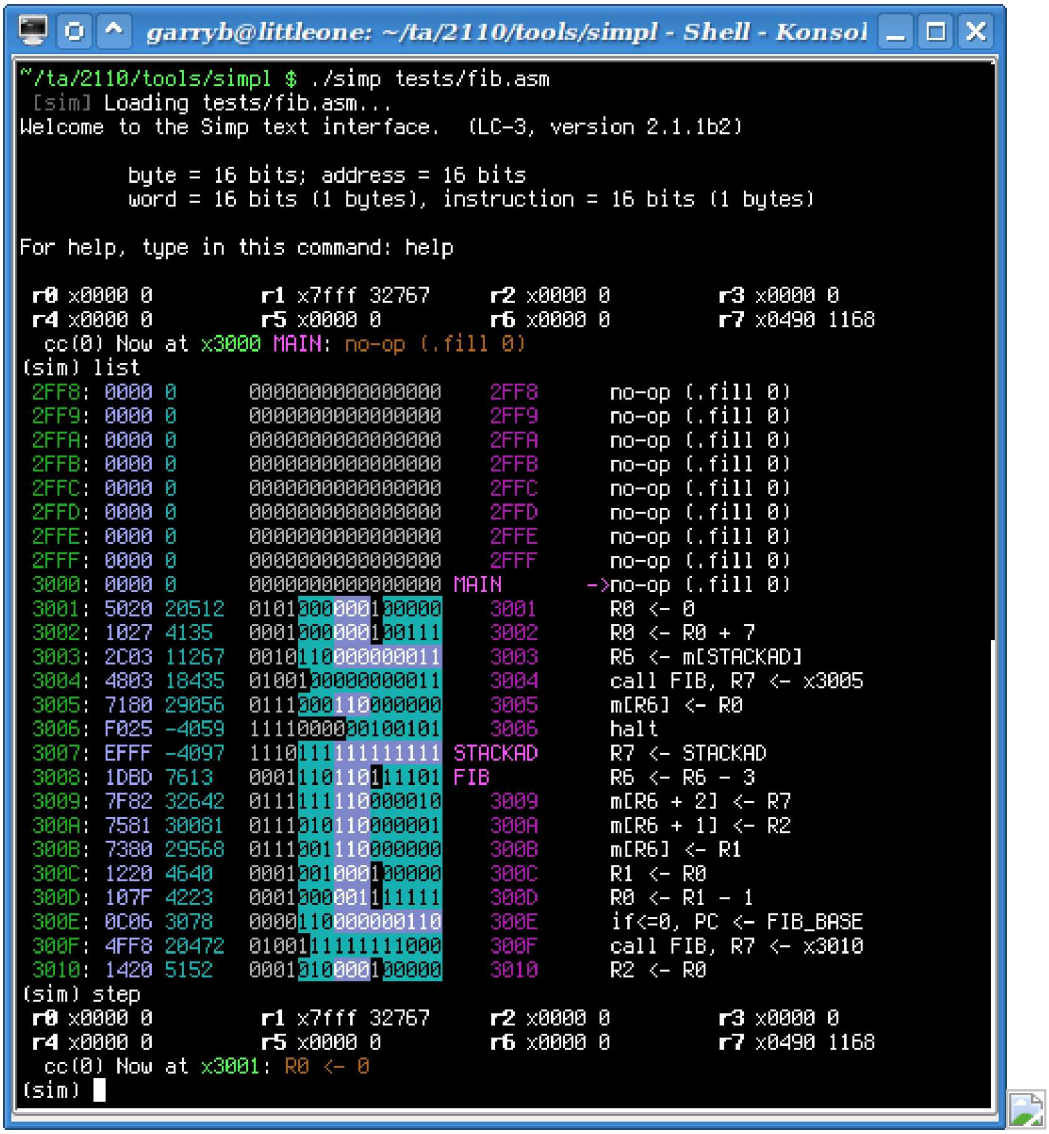
At a quick glance, here are features of interest for SimpLC.

Feature	Status
General	
Text mode (like GDB)	Yes
Graphical mode	Yes
Load assembly/bin/hex files directly	Yes
Native	Yes
Fast and responsive	Yes
Integrated editor	Use your favorite editor
Cycle-accuracy, device emulation	No - our focus is on beginners
Project system	No - our focus is on beginners
Simulated Architectures	
LC-3	Yes
LC-2200	Yes
Brain*	Yes
MIPS	In progress
I/O	
Blocking or polling console	Yes
Graphical framebuffer	In consideration
Interrupts	In consideration
Platforms	
Linux	Yes
Windows	Yes
Mac	Yes
Source code	
Open source	Yes (BSD license)
Language	C++

Modular	Yes

The Text Mode Simulator

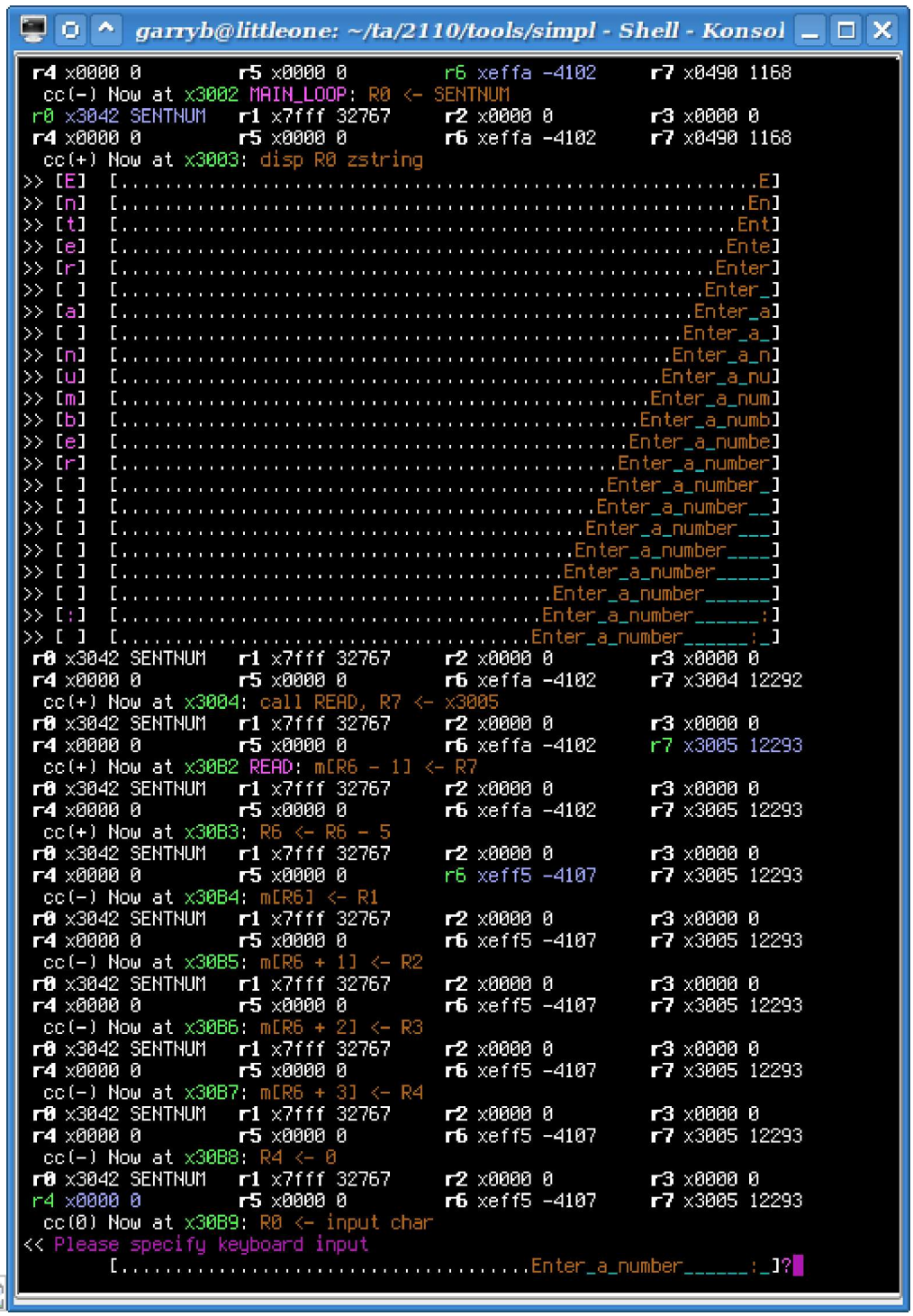
The Basic Text Interface



This is the text interface for the simulator. As you can see, it is quite colorful. However, the color is not simply for prettiness; it aids in emphasizing consistency among various views. Addresses are green; dark blue is almost always a hexadecimal value, whereas light blue is decimal, and so on. However, the most obvious use of color is to separate each operand of the binary view. This is done automatically for all ISA's that use the powerful built-in assembler framework.

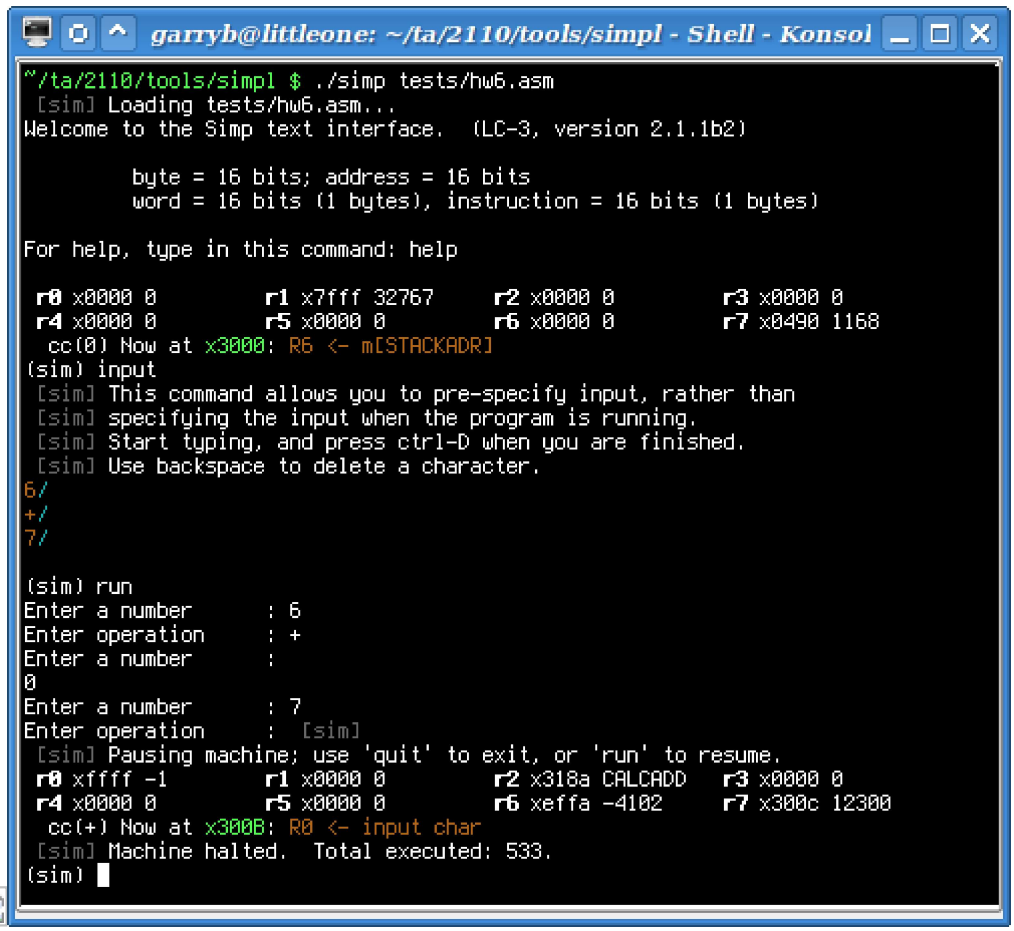
If you look at the first couple commands, the user is trying to set the instruction at x3007 to an assembly instruction. The user is trying to test R0 for the control code, but accidentally zeros it out. The symbolic disassembly makes this obvious by displaying "R0 <- 0", and the user can quickly change the instruction to what was intended.

Verbose Mode



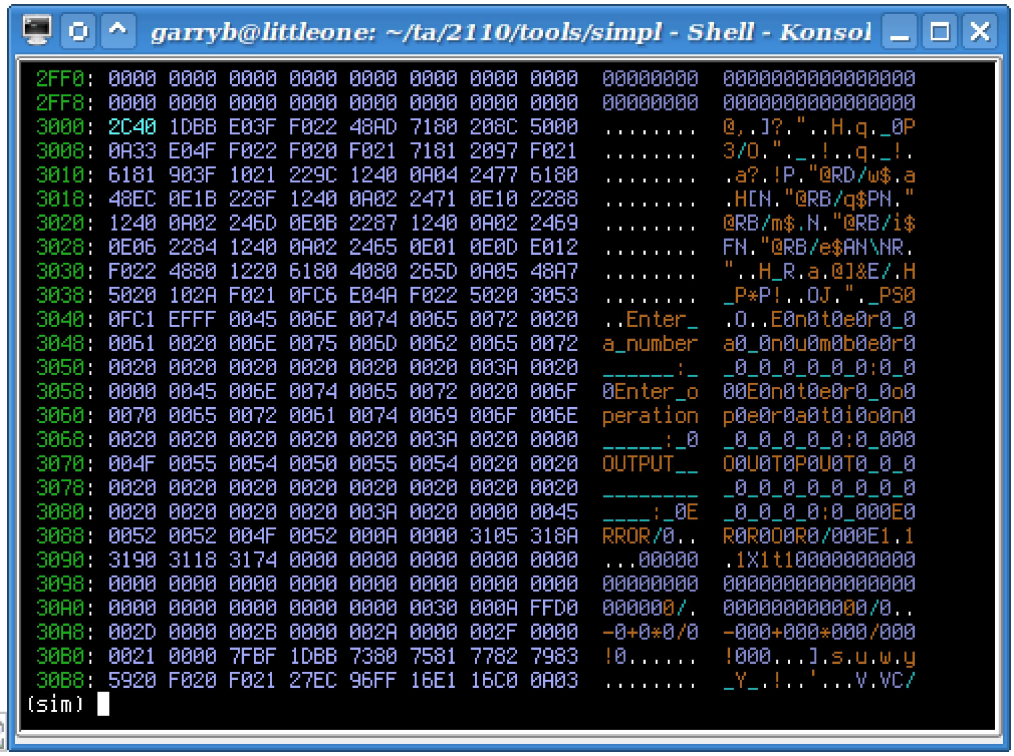
This shows the simulator in action, with the verbose mode enabled. In verbose mode, every gruesome detail about execution is printed. Care is taken so that verbose mode interacts cleanly with input/output, so that any relevant prompts are displayed when input is requested. Of course, in non-verbose mode, the simulator does not display such prolific output.

Canning Input



This is the interface for pre-specifying canned input to be handled later. ASCII characters are highlighted orange like usual and whitespace is highlighted light blue. In addition to the `input` command, there is also the `infile` command that reads input from a file into the input buffer.

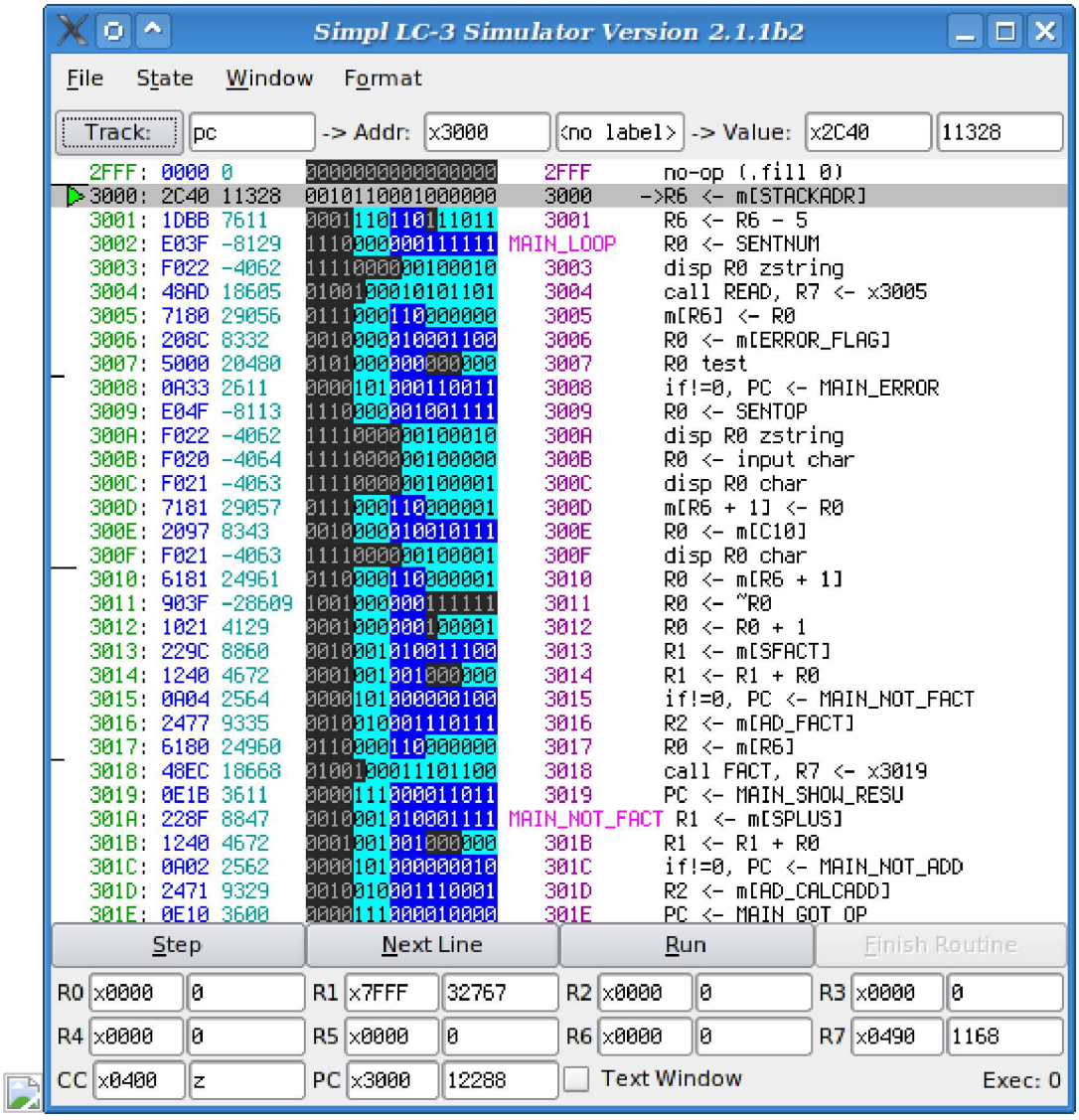
Compact Byte View



Like many simulators, this simulator provides a compact view of memory in much the way hexdump does. The green columns specify address, whereas the blue numbers are the bytes. The last two columns are text views of memory at the 16-bit (word) levels, and also at half-word levels for compact strings. If the color is a bit too much, the command "viewm -k" will nicely disable colors.

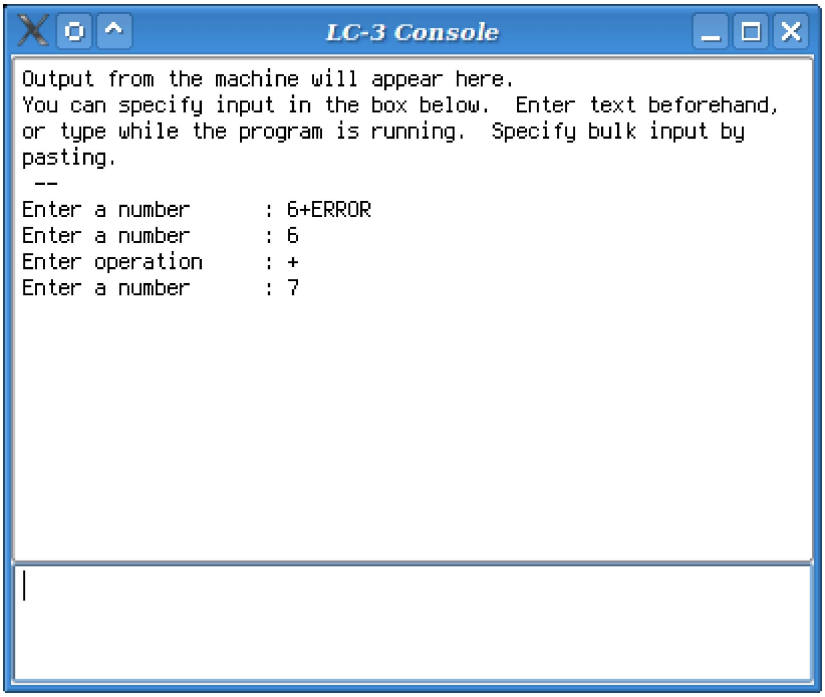
The GUI Simulator

Basic Graphical Interface



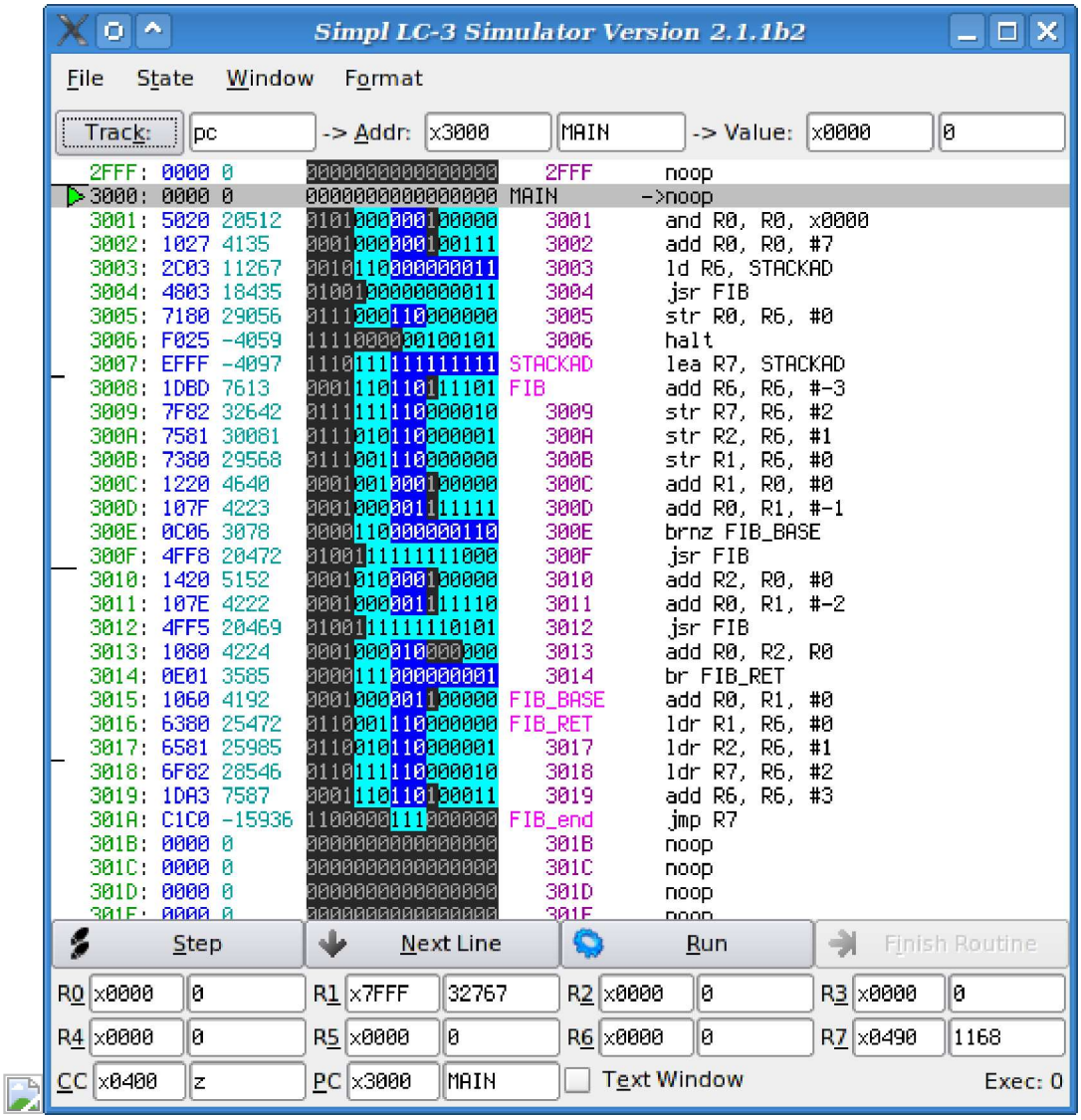
This is the basic graphical interface. It looks quite similar to the text interface in its use of colors. At this point in execution, the simulator is waiting on user input. On the top is a widget that lets you track various expressions as a memory address, and change memory values. On the bottom are controls for execution and for modifying registers. Like in the text interface, expressions are allowed to be complex, including arithmetic, register names, and dereferencing.

The GUI Console



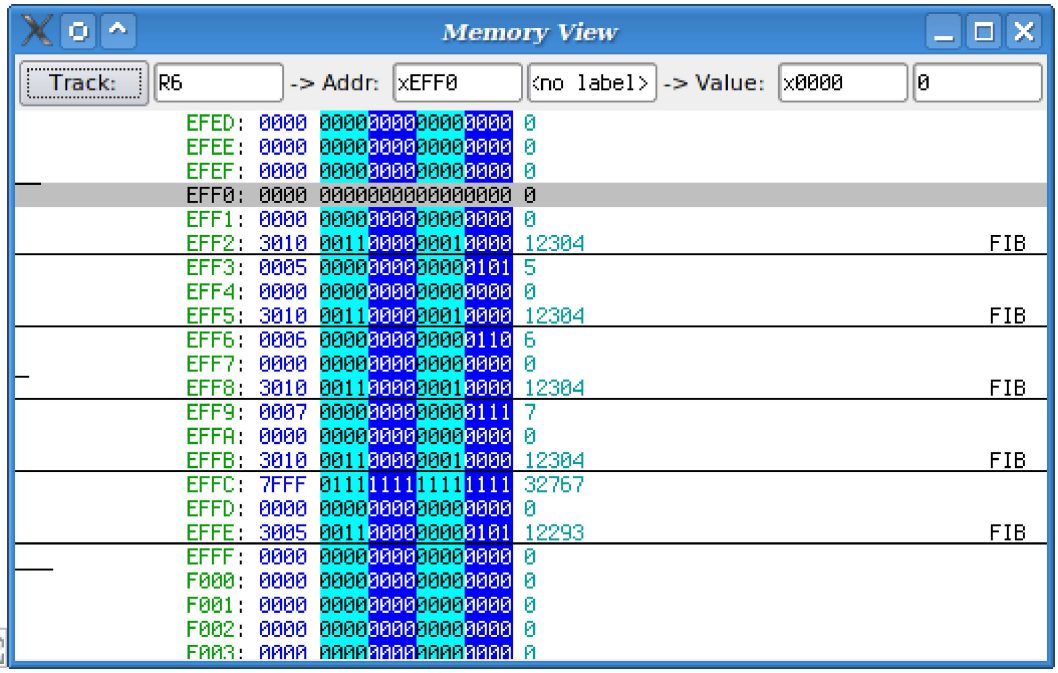
This is the console interface to the graphical simulator. On the top is all the output generated by the program. Below is a text box where you can interactively type input, or paste canned input.

A Fibonacci Program



Here, a stack-based Fibonacci program is shown in motion. Here, plain disassembly instead of symbolic disassembly is shown. (For interested developers: Once you define the assembly language, the disassembly requires zero extra code.) The simulator automatically tracks a call trace and thus offers an option to finish the current instance of FIB.

The Fibonacci Stack



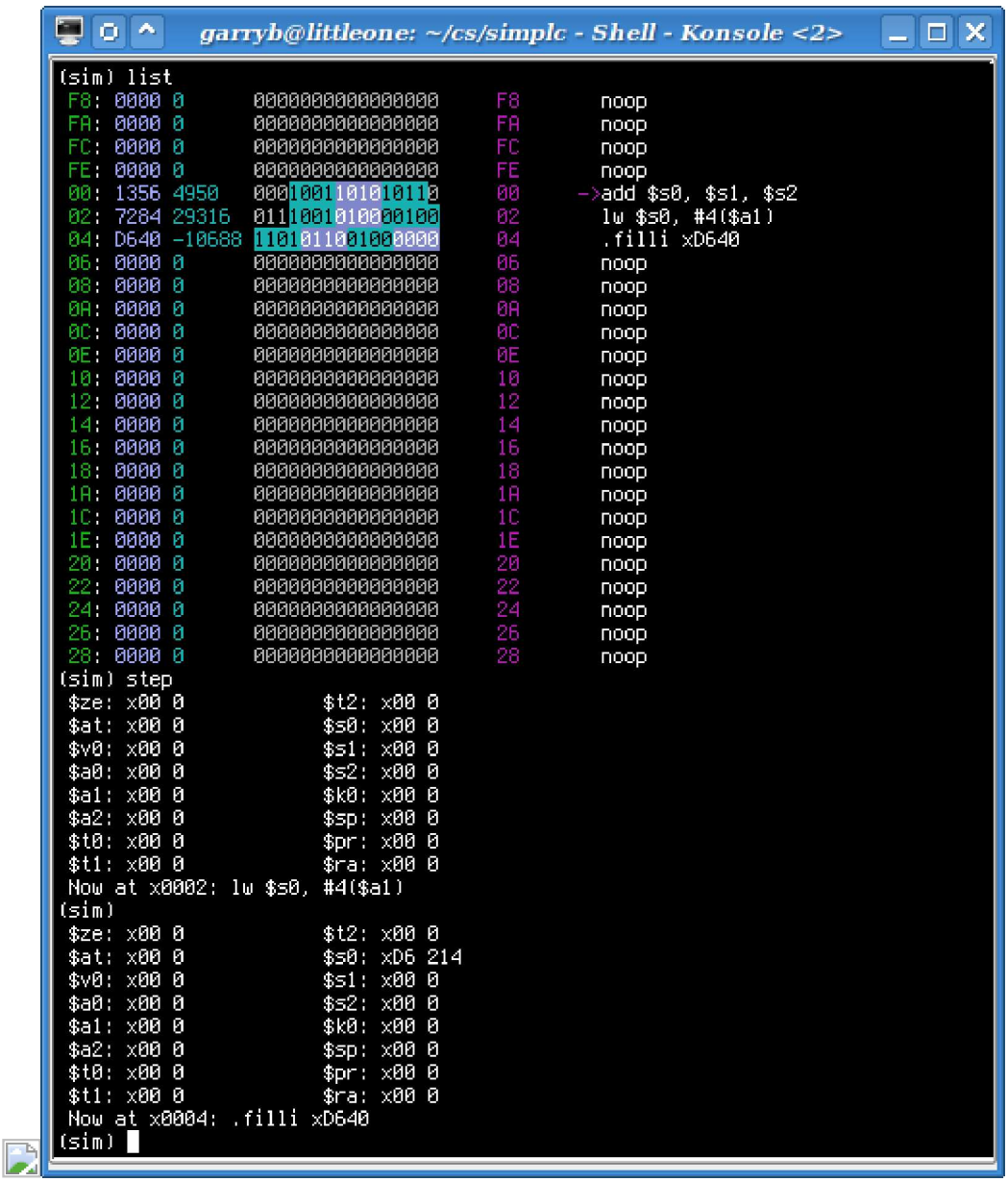
This is perhaps one of the most fun things about this simulator. A user can have any number of alternate memory views in addition to the main view; the code window and stack window are running side by side. In the "Track" text box, the value of register 6 is currently being tracked. The stack frames are automatically recognized and sectioned off for each instance of FIB. To watch the stack jump up and down, hold the F11 key in any window.

In MS Windows

And yes, there is a full-featured Windows version! (Screenshot needed)

ISA Extensibility

View of (the old) LC-2200-8



LC-2200-8 (GT8) is another architecture it supports, which is used at Georgia Tech for an introductory systems class. In this architecture, bytes and words are 8 bits, but instructions are 16 bits. Note how instruction disassembly works just the same and the data list views words in the right size. Finally, unlike the dump view in LC-3 with 16 bits in a single byte, 16 bytes (not 8) are fit onto a line, and the packed-string view is omitted since it does not apply to 8-bit-byte systems.

The on-the-fly assembly and other features all work in this version of the simulator too. Extra architectures are modeled by objects, in a way that it is possible to run two different types of machines at the same time (although I am not sure why you would want to).

Defining an Assembly Language

No kidding, this is the entire assembler for a LC-2200:

```
AsmRuleGenerator gen;

// register names
mRegTable.add("$zero", 0).add("$at", 1) .add("$v0", 2) .add("$a0", 3);
mRegTable.add("$a1", 4)  .add("$a2", 5) .add("$t0", 6) .add("$t1", 7);
mRegTable.add("$t2", 8)  .add("$s0", 9) .add("$s1", 10).add("$s2", 11);
mRegTable.add("$k0", 12) .add("$sp", 13).add("$pr", 14).add("$ra", 15);

// bind operator types, that we refer to later
gen.bind("regX", new TableRule(&mRegTable, 9, 4));
```

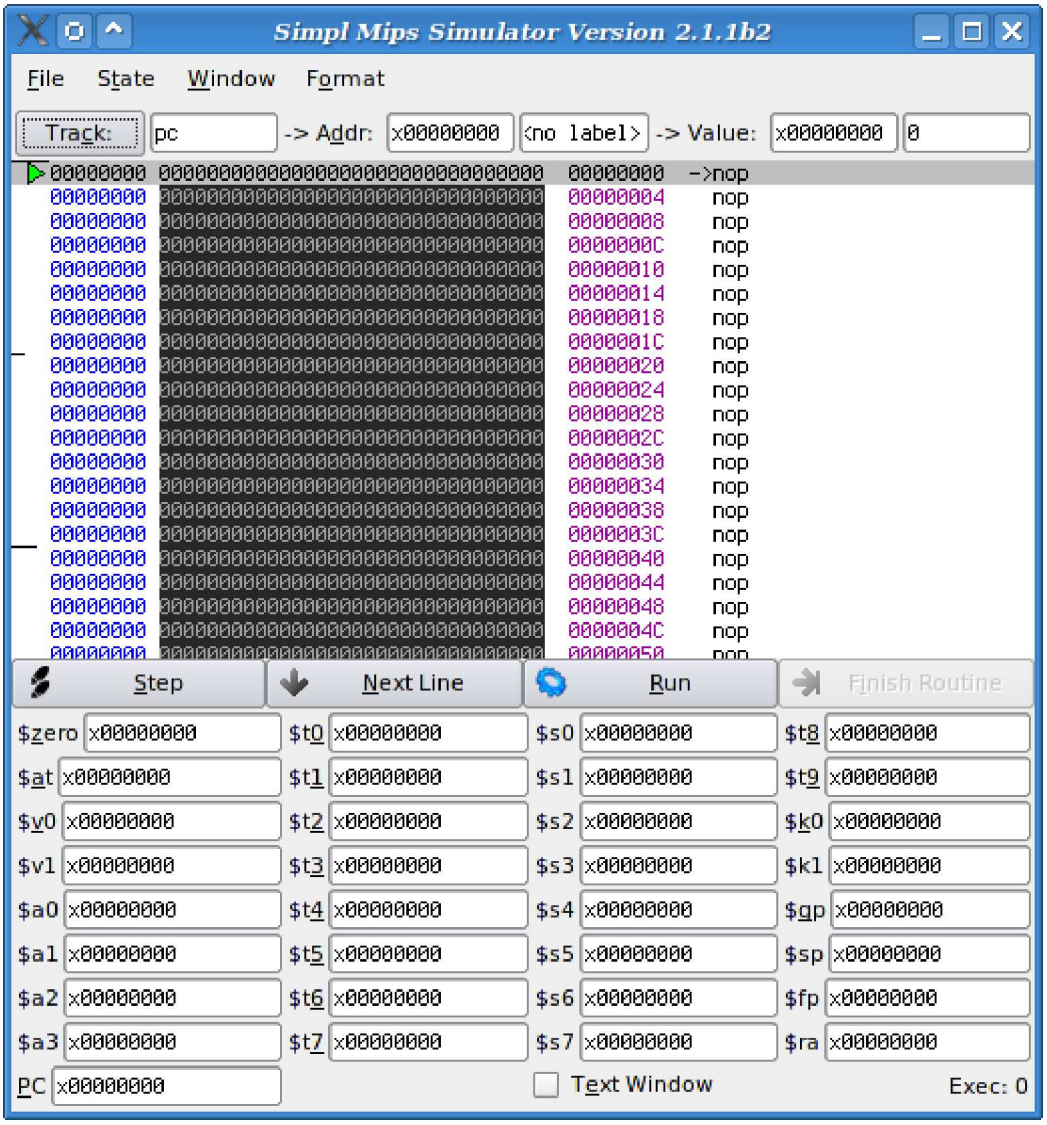
```
gen.bind("regY", new TableRule(&mRegTable, 5, 4));
gen.bind("regZ", new TableRule(&mRegTable, 1, 4));
gen.bind("imm2u",
    new ImmRule(2, Format::DisplayDec, ImmRule::Unsigned, 0, 2));
gen.bind("imm5",
    new ImmRule(16, Format::DisplayDec, ImmRule::Signed, 0, 5));
gen.bind("imm8",
    new ImmRule(16, Format::DisplayDec, ImmRule::Signed, 0, 5));
gen.bind("pcoff5",
    new ImmRule(16, Format::DisplayHexSym,
        ImmRule::Signed|ImmRule::PCOff, 0, 5, 1, 1));

// instructions, using previously defined operator types
gen.add("0000 0000 0000 0000", "noop");
gen.add("000. .... .... ....", "add regX,regY,regZ");
gen.add("001. .... .... ....", "nand regX,regY,regZ");
gen.add("010. .... .... ....", "addi regX,regY,imm5");
gen.add("011. .... .... ....", "lw regX,imm8(regY)");
gen.add("100. .... .... ....", "sw regX,imm8(regY)");
gen.add("101. .... .... ....", "beq regX,regY,pcoff5");
gen.add("110. .... ...0 0000", "jalr regX,regY");
gen.add("1110 0000 0000 0000", "halt");
gen.add("1110 0000 0000 00..", "spop imm2u");

setAsmRule(gen.extractRoot());
```

A Preview of MIPS...

Here's a preview of MIPS. Unfortunately, the number of registers make an unsightly view. To make up for having a whopping 32 bits, some fields are removed on the disassembly view.



Comments

You do not have permission to add comments.