

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++

The Text Mode Simulator

The Basic Text Interface

```

~/ta/2110/tools/simpl $ ./simp tests/fib.asm
[sim] Loading tests/fib.asm...
Welcome to the Simpl text interface. (LC-3, version 2.1.1b2)

    byte = 16 bits; address = 16 bits
    word = 16 bits (1 bytes), instruction = 16 bits (1 bytes)

For help, type in this command: help

r0 x0000 0      r1 x7fff 32767      r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 x0000 0      r7 x0490 1168
cc(0) Now at x3000 MAIN: no-op (.fill 0)
(sim) list
2FF8: 0000 0      0000000000000000      2FF8      no-op (.fill 0)
2FF9: 0000 0      0000000000000000      2FF9      no-op (.fill 0)
2FFA: 0000 0      0000000000000000      2FFA      no-op (.fill 0)
2FFB: 0000 0      0000000000000000      2FFB      no-op (.fill 0)
2FFC: 0000 0      0000000000000000      2FFC      no-op (.fill 0)
2FFD: 0000 0      0000000000000000      2FFD      no-op (.fill 0)
2FFE: 0000 0      0000000000000000      2FFE      no-op (.fill 0)
2FFF: 0000 0      0000000000000000      2FFF      no-op (.fill 0)
3000: 0000 0      0000000000000000      MAIN      ->no-op (.fill 0)
3001: 5020 20512  0101000001000000      3001      R0 <- 0
3002: 1027 4135   0001000000100111      3002      R0 <- R0 + 7
3003: 2C83 11267  0010110000000011      3003      R6 <- m[STACKAD]
3004: 4883 18435  0100100000000011      3004      call FIB, R7 <- x3005
3005: 7180 29056  0111000110000000      3005      m[R6] <- R0
3006: F025 -4059  1111000000100101      3006      halt
3007: EFFF -4097  1110111111111111      STACKAD  R7 <- STACKAD
3008: 10BD 7613   0001100100111001      FIB      R6 <- R6 - 3
3009: 7F82 32642  0111111000000010      3009      m[R6 + 2] <- R7
300A: 7581 30081  0111001010000001      300A      m[R6 + 1] <- R2
300B: 7380 29568  0111001110000000      300B      m[R6] <- R1
300C: 1220 4640   0001000100010000      300C      R1 <- R0
300D: 107F 4223   0001000000111111      300D      R0 <- R1 - 1
300E: 0C06 3078   0000110000000010      300E      if<=0, PC <- FIB_BASE
300F: 4FF8 20472  0100111111111100      300F      call FIB, R7 <- x3010
3010: 1420 5152   0001000001000000      3010      R2 <- R0

(sim) step
r0 x0000 0      r1 x7fff 32767      r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 x0000 0      r7 x0490 1168
cc(0) Now at x3001: R0 <- 0
(sim) 
```

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

```

r4 x0000 0      r5 x0000 0      r6 xeffa -4102    r7 x0490 1168
cc(-) Now at x3002 MAIN_LOOP: R0 <- SENTHUM
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeffa -4102    r7 x0490 1168
cc(+) Now at x3003: disp R0 zstring
>> [E] [.....] .E]
>> [n] [.....] .En]
>> [t] [.....] .Ent]
>> [e] [.....] .Ente]
>> [r] [.....] .Enter]
>> [ ] [.....] .Enter_]
>> [a] [.....] .Enter_a]
>> [ ] [.....] .Enter_a_]
>> [n] [.....] .Enter_a_n]
>> [u] [.....] .Enter_a_nu]
>> [m] [.....] .Enter_a_num]
>> [b] [.....] .Enter_a_numb]
>> [e] [.....] .Enter_a_nume]
>> [r] [.....] .Enter_a_number]
>> [ ] [.....] .Enter_a_number_]
>> [ ] [.....] .Enter_a_number__]
>> [ ] [.....] .Enter_a_number___]
>> [ ] [.....] .Enter_a_number____]
>> [ ] [.....] .Enter_a_number_____]
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeffa -4102    r7 x3004 12292
cc(+) Now at x3004: call READ, R7 <- x3005
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeffa -4102    r7 x3005 12293
cc(+) Now at x30B2 READ: mCR6 - 11 <- R7
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeffa -4102    r7 x3005 12293
cc(+) Now at x30B3: R6 <- R6 - 5
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeff5 -4107    r7 x3005 12293
cc(-) Now at x30B4: mCR6] <- R1
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeff5 -4107    r7 x3005 12293
cc(-) Now at x30B5: mCR6 + 11 <- R2
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeff5 -4107    r7 x3005 12293
cc(-) Now at x30B6: mCR6 + 2] <- R3
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeff5 -4107    r7 x3005 12293
cc(-) Now at x30B7: mCR6 + 3] <- R4
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeff5 -4107    r7 x3005 12293
cc(-) Now at x30B8: R4 <- 0
r0 x3042 SENTHUM   r1 x7fff 32767   r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeff5 -4107    r7 x3005 12293
cc(0) Now at x30B9: R0 <- input char
<< Please specify keyboard input
[.....] .Enter_a_number_____]?]
```

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

```
"/ta/2110/tools/simpl $ ./simp tests/hw6.asm
[Sim] Loading tests/hw6.asm...
Welcome to the Simp text interface. (LC-3, version 2.1.1b2)

    byte = 16 bits; address = 16 bits
    word = 16 bits (1 bytes), instruction = 16 bits (1 bytes)

For help, type in this command: help

r0 x0000 0      r1 x7fff 32767      r2 x0000 0      r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 x0000 0      r7 x0490 1168
cc(0) Now at x3000: R6 <- m[STACKADR]
(sim) input
[Sim] This command allows you to pre-specify input, rather than
[Sim] specifying the input when the program is running.
[Sim] Start typing, and press ctrl-D when you are finished.
[Sim] Use backspace to delete a character.

6/
+/
7/

(sim) run
Enter a number : 6
Enter operation : +
Enter a number :
0
Enter a number : 7
Enter operation : [sim]
[Sim] Pausing machine; use 'quit' to exit, or 'run' to resume.
r0xffff -1      r1 x0000 0      r2 x318a CALCADD  r3 x0000 0
r4 x0000 0      r5 x0000 0      r6 xeffa -4102     r7 x300c 12300
cc(+) Now at x300B: R0 <- input char
[Sim] Machine halted. Total executed: 533.
(sim) █
```

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

```
2FF0: 0000 0000 0000 0000 0000 0000 0000 0000 00000000 0000000000000000
2FF8: 0000 0000 0000 0000 0000 0000 0000 0000 00000000 0000000000000000
3000: 2C40 1DBB E03F F022 48AD 7180 208C 5000 .....
3008: 0A33 E04F F022 F020 F021 7181 2097 F021 .....
3010: 6181 903F 1021 229C 1240 0A04 2477 6180 .....
3018: 48EC 0E18 228F 1240 0A02 2471 0E10 2288 .....
3020: 1240 0A02 246D 0E0B 2287 1240 0A02 2469 .....
3028: 0E06 2284 1240 0A02 2465 0E01 0E0D E012 .....
3030: F022 4880 1220 6180 4080 265D 0A05 48A7 .....
3038: 5020 102A F021 0FC6 E04A F022 5020 3053 .....
3040: 0FC1 EFFF 0045 006E 0074 0065 0072 0020 ..Enter_.
3048: 0061 0020 006E 0075 006D 0062 0065 0072 a_number a0_0n0U0m0b0r0
3050: 0020 0020 0020 0020 0020 003A 0020 0020 .....
3058: 0000 0045 006E 0074 0065 0072 0020 006F 0Enter_o 00E0n0l0e0r0.0o
3060: 0070 0065 0072 0061 0074 0069 006F 006E peration p0e0-0a0t0i0o0n0
3068: 0020 0020 0020 0020 0020 003A 0020 0000 .....
3070: 004F 0055 0054 0050 0055 0054 0020 0020 OUTPUT_ 00U0T0P0U0T0_0_
3078: 0020 0020 0020 0020 0020 0020 0020 0020 .....
3080: 0020 0020 0020 0020 003A 0020 0000 0045 .....
3088: 0052 0052 004F 0052 000A 0000 3105 318A RROR/0.. R0R000R0/000E1.1
3090: 3190 3118 3174 0000 0000 0000 0000 0000 .1X1t100000000000
3098: 0000 0000 0000 0000 0000 0000 0000 0000 00000000 0000000000000000
30A0: 0000 0000 0000 0000 0030 000A FFD0 000000/. 000000000000/0..
30A8: 002D 0000 002B 0000 002A 0000 002F 0000 -0+0*0/0 -000+000*000/000
30B0: 0021 0000 7FBF 1DBB 7380 7581 7782 7983 !0..... !000...].s.u.w.y
30B8: 5920 F020 F021 27EC 96FF 16E1 16C0 0A03 ..... Y_.!...'V.VC/
(sim) █
```

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

The screenshot shows the Simpl LC-3 Simulator Version 2.1.1b2 interface. At the top, there's a menu bar with File, State, Window, and Format. Below the menu is a toolbar with a track selection button, a PC entry field (pc), an address entry field (Addr: x3000), a label entry field (no label), a value entry field (Value: x2C40), and a numeric entry field (11328). The main window displays assembly code in a table format. The columns are Address (green), Value (blue), and Disassembly (purple). The assembly code is as follows:

2FFF: 0000 0	0000000000000000	2FFF no-op (.fill 0)
3000: 2C40 11328	0010110000100000	3000 ->R6 <- m[STACKADR]
3001: 1DBB 7611	0001110111011101	3001 R6 <- R6 - 5
3002: E03F -8129	1110000000111111	3002 R0 <- SENTNUM
3003: F022 -4062	1111000000100000	3003 disp R0 zstring
3004: 48AD 18605	0100100010101101	3004 call READ, R7 <- x3005
3005: 7180 29056	0111000011000000	3005 m[R6] <- R0
3006: 208C 8332	0010000001000100	3006 R0 <- m[ERROR_FLAG]
3007: 5000 20480	0101000000000000	3007 R0 test
3008: 0A33 2611	0000100011000000	3008 if!=0, PC <- MAIN_ERROR
3009: E04F -8113	1110000000100000	3009 R0 <- SENTOP
300A: F022 -4062	1111000000010000	300A disp R0 zstring
300B: F020 -4064	1111000000010000	300B R0 <- input char
300C: F021 -4063	1111000000010001	300C disp R0 char
300D: 7181 29057	0111000011000000	300D m[R6 + 1] <- R0
300E: 2097 8343	0010000001000100	300E R0 <- m[C10]
300F: F021 -4063	1111000000010001	300F disp R0 char
3010: 6181 24961	0110000000100001	3010 R0 <- m[R6 + 1]
3011: 903F -28609	1001000000111111	3011 R0 <- "R0"
3012: 1021 4129	0001000000100001	3012 R0 <- R0 + 1
3013: 229C 8860	0010001010001100	3013 R1 <- m[SFACT]
3014: 1240 4672	0001001001000000	3014 R1 <- R1 + R0
3015: 0A04 2564	0000100100000000	3015 if!=0, PC <- MAIN_NOT_FACT
3016: 2477 9335	0010001001110000	3016 R2 <- m[AD_FACT]
3017: 6180 24960	0110000000110000	3017 R0 <- m[R6]
3018: 48EC 18668	0100010001110100	3018 call FACT, R7 <- x3019
3019: 0E1B 3611	0000111000001101	3019 PC <- MAIN_SHOW_RESU
301A: 228F 8847	0010001000000000	301A MAIN_NOT_FACT R1 <- m[SPPLUS]
301B: 1240 4672	0001001001000000	301B R1 <- R1 + R0
301C: 0A02 2562	0000100100000000	301C if!=0, PC <- MAIN_NOT_ADD
301D: 2471 9329	0010001000111000	301D R2 <- m[AD_CALCADD]
301E: 0E10 3600	0000111000000000	301E PC <- MAIN GOT_OP

Below the assembly table are four buttons: Step, Next Line, Run, and Finish Routine. Underneath these buttons are four rows of register controls. The first row contains R0 (x0000, 0), R1 (x7FFF, 32767), R2 (x0000, 0), R3 (x0000, 0). The second row contains R4 (x0000, 0), R5 (x0000, 0), R6 (x0000, 0), R7 (x0490, 1168). The third row contains CC (x0400, z), PC (x3000, 12288), and a checkbox for Text Window. The fourth row contains Exec: 0.

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

The screenshot shows a Windows-style application window titled "LC-3 Console". The title bar has standard minimize, maximize, and close buttons. The main area contains text output and an input field.

Output from the machine will appear here.
You can specify input in the box below. Enter text beforehand,
or type while the program is running. Specify bulk input by
pasting.

--

```
Enter a number      : 6+ERROR
Enter a number      : 6
Enter operation     : +
Enter a number      : 7
```

A vertical cursor bar is visible in the input field at the bottom left.

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

Simpl LC-3 Simulator Version 2.1.1b2

File State Window Format

Track:	pc	-> Addr: x3000	MAIN	-> Value: x0000	0
--------	----	----------------	------	-----------------	---

```

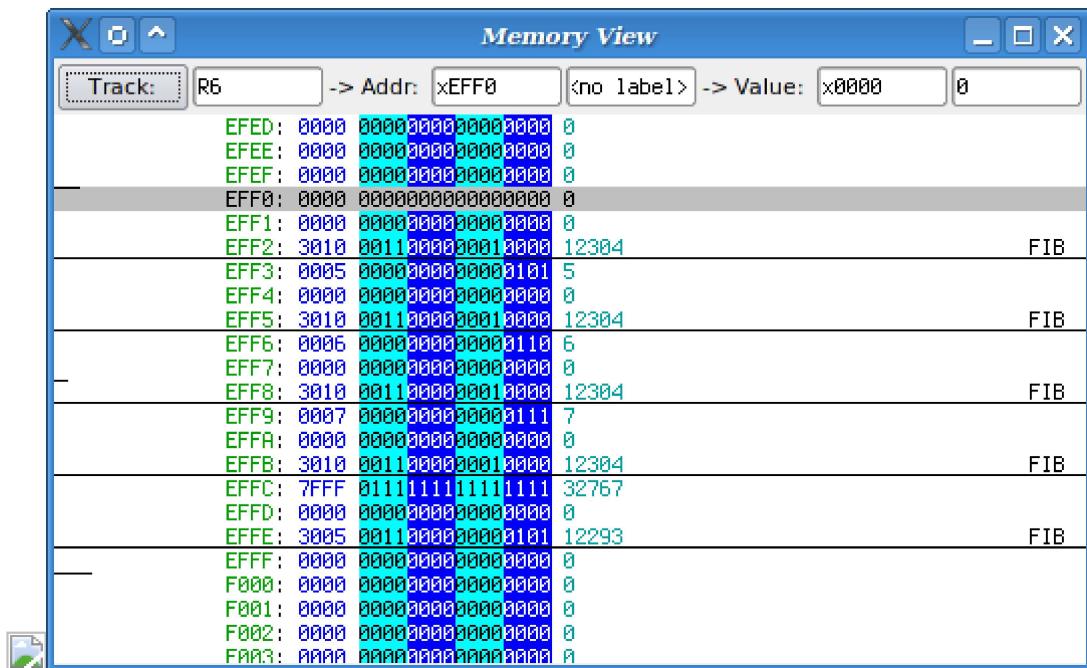
2FFF: 0000 0 0000000000000000 2FFF noop
>3000: 0000 0 0000000000000000 MAIN ->noop
3001: 5020 20512 0101000000100000 3001 and R0, R0, x0000
3002: 1027 4135 0001000001001111 3002 add R0, R0, #7
3003: 2003 11267 0010110000000011 3003 ldr R6, STACKAD
3004: 4803 18435 0100100000000111 3004 jsr FIB
3005: 7180 29056 0111000110000000 3005 str R0, R6, #0
3006: F025 -4059 1111000000100101 3006 halt
3007: EFFF -4097 1110111111111111 STACKAD 3007 lea R7, STACKAD
3008: 1D6D 7613 0001110110111101 FIB 3008 add R6, R6, #-3
3009: 7F82 32642 0111111110000010 3009 str R7, R6, #2
300A: 7581 30081 0111010110000001 300A str R2, R6, #1
300B: 7380 29568 0111001110000000 300B str R1, R6, #0
300C: 1220 4640 0001001000100000 300C add R1, R0, #0
300D: 107F 4223 0001000001111111 300D add R0, R1, #-1
300E: 0C06 3078 0000110000000110 300E brnz FIB_BASE
300F: 4FF8 20472 0100111111111100 300F jsr FIB
3010: 1420 5152 0001310000100000 3010 add R2, R0, #0
3011: 107E 4222 0001000001111110 3011 add R0, R1, #-2
3012: 4FF5 20469 0100111111110101 3012 jsr FIB
3013: 1080 4224 0001000010000000 3013 add R0, R2, R0
3014: 0E01 3585 0000111000000001 3014 br FIB_RET
3015: 1060 4192 0001000001100000 FIB_BASE 3015 add R0, R1, #0
3016: 6380 25472 0110001110000000 FIB_RET 3016 ldr R1, R6, #0
3017: 6581 25985 0110310110000001 3017 ldr R2, R6, #1
3018: 6F82 28546 0110111100000010 3018 ldr R7, R6, #2
3019: 1DA3 7587 0001110110010011 3019 add R6, R6, #3
301A: C1C0 -15936 1100000111000000 FIB_end 301A jmp R7
301B: 0000 0 0000000000000000 301B noop
301C: 0000 0 0000000000000000 301C noop
301D: 0000 0 0000000000000000 301D noop
301E: 0000 0 0000000000000000 301E noop

```

	Step		Next Line		Run		Finish Routine				
R0	x0000	0	R1	x7FFF	32767	R2	x0000	0	R3	x0000	0
R4	x0000	0	R5	x0000	0	R6	x0000	0	R7	x0490	1168
CC	x0400	z	PC	x3000	MAIN	<input type="checkbox"/> Text Window	Exec: 0				

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



The screenshot shows a 'Memory View' window with the title bar 'Memory View'. At the top, there are four input fields: 'Track:' (containing 'R6'), 'Addr:' (containing 'xEFF0'), 'Label:' (containing '<no label>'), and 'Value:' (containing '0'). Below these fields is a table of memory addresses and their values. The table has columns for Address (e.g., EFFE), Value (e.g., 0000), and Type (e.g., FIB). The rows are color-coded by address range: green for EFEF-EFF0, blue for EFF1-EFF2, red for EFF3-EFF4, orange for EFF5-EFF6, yellow for EFF7-EFF8, purple for EFF9-EFFA, pink for EFFB-EFFC, light blue for EFFD-EFFE, and grey for EFFF-F003. The 'FIB' label is present in several rows, indicating function entry points.

Address	Value	Type
EFED: 0000	0000000000000000	0
EFEE: 0000	0000000000000000	0
EFFE: 0000	0000000000000000	0
EFF0: 0000	0000000000000000	0
EFF1: 0000	0000000000000000	0
EFF2: 3010 0011000000010000	12304	FIB
EFF3: 0005	0000000000000101	5
EFF4: 0000	0000000000000000	0
EFF5: 3010 0011000000010000	12304	FIB
EFF6: 0006	0000000000000110	6
EFF7: 0000	0000000000000000	0
EFF8: 3010 0011000000010000	12304	FIB
EFF9: 0007	0000000000000111	7
EFFA: 0000	0000000000000000	0
EFFB: 3010 0011000000010000	12304	FIB
EFFC: 7FFF 0111111111111111	32767	
EFFD: 0000	0000000000000000	0
EFFE: 3005 0011000000000101	12293	FIB
EFFF: 0000	0000000000000000	0
F000: 0000	0000000000000000	0
F001: 0000	0000000000000000	0
F002: 0000	0000000000000000	0
F003: 0000 0000000000000000	0	

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

```

(garryb@littleone: ~/cs/simple - Shell - Konsole <2>)
(sim) list
F8: 0000 0    0000000000000000    F8    noop
FA: 0000 0    0000000000000000    FA    noop
FC: 0000 0    0000000000000000    FC    noop
FE: 0000 0    0000000000000000    FE    noop
00: 1356 4950  00100101010110    00    ->add $s0, $s1, $s2
02: 7284 29316  01110010100000100   02    lw $s0, #4($a1)
04: D640 -10688  1101011001000000   04    .filli xD640
06: 0000 0    0000000000000000    06    noop
08: 0000 0    0000000000000000    08    noop
0A: 0000 0    0000000000000000    0A    noop
0C: 0000 0    0000000000000000    0C    noop
0E: 0000 0    0000000000000000    0E    noop
10: 0000 0    0000000000000000    10    noop
12: 0000 0    0000000000000000    12    noop
14: 0000 0    0000000000000000    14    noop
16: 0000 0    0000000000000000    16    noop
18: 0000 0    0000000000000000    18    noop
1A: 0000 0    0000000000000000    1A    noop
1C: 0000 0    0000000000000000    1C    noop
1E: 0000 0    0000000000000000    1E    noop
20: 0000 0    0000000000000000    20    noop
22: 0000 0    0000000000000000    22    noop
24: 0000 0    0000000000000000    24    noop
26: 0000 0    0000000000000000    26    noop
28: 0000 0    0000000000000000    28    noop
(sim) step
$ze: x00 0    $t2: x00 0
$at: x00 0    $s0: x00 0
$v0: x00 0    $s1: x00 0
$a0: x00 0    $s2: x00 0
$a1: x00 0    $k0: x00 0
$a2: x00 0    $sp: x00 0
$t0: x00 0    $pr: x00 0
$t1: x00 0    $ra: x00 0
Now at x0002: lw $s0, #4($a1)
(sim)
$ze: x00 0    $t2: x00 0
$at: x00 0    $s0: x06 214
$v0: x00 0    $s1: x00 0
$a0: x00 0    $s2: x00 0
$a1: x00 0    $k0: x00 0
$a2: x00 0    $sp: x00 0
$t0: x00 0    $pr: x00 0
$t1: x00 0    $ra: x00 0
Now at x0004: .filli xD640
(sim)

```

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));

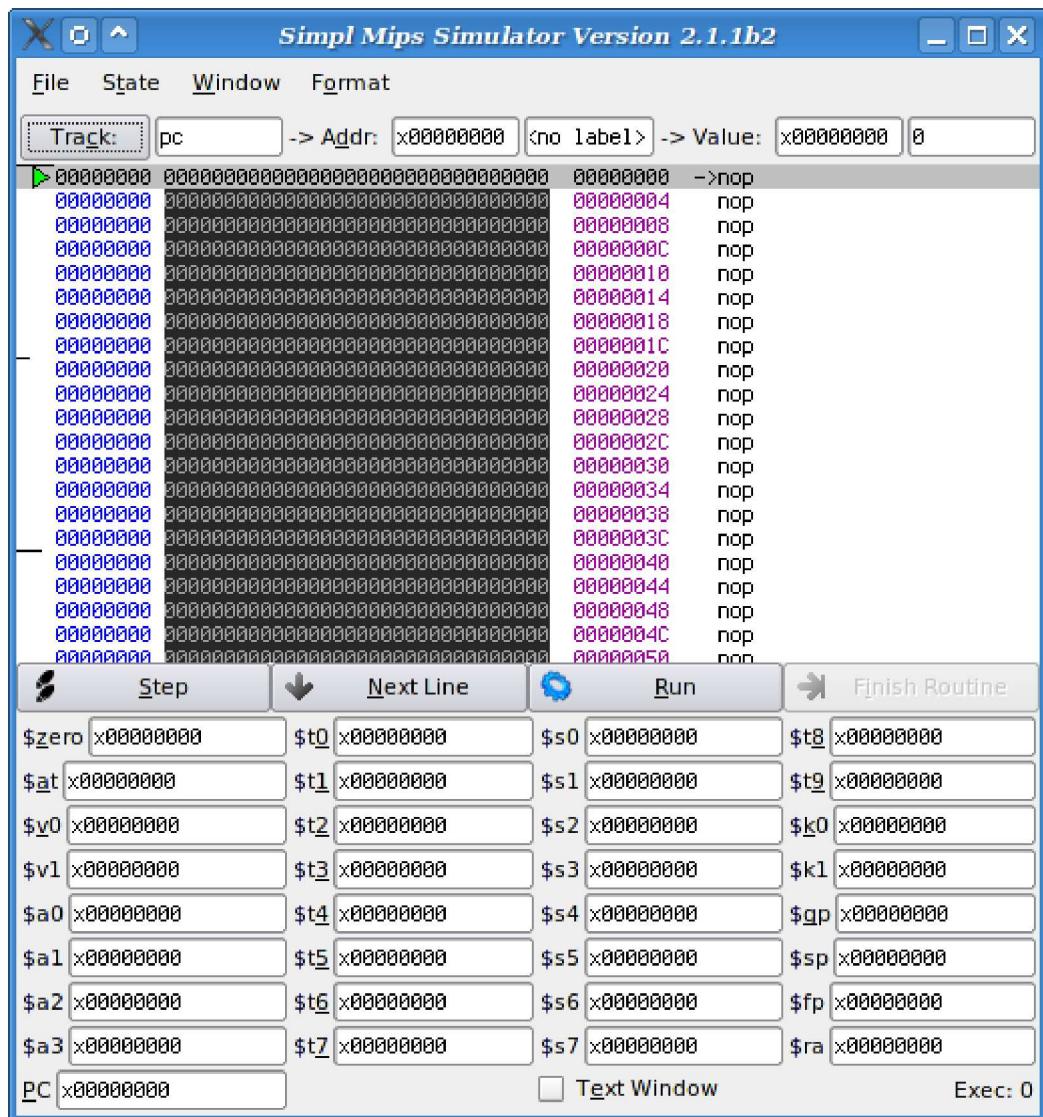
// instructions, using previously defined operator types
gen.add("0000 0000 0000 0000", "noop");
gen.add("000. .... .... ....", "add regX,regY,regZ");
gen.add("001. .... .... ....", "nand regX,regY,regZ");
gen.add("010. .... .... ....", "addi regX,regY,imm5");
gen.add("011. .... .... ....", "lw regX,imm8(regY)");
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.