Lovejit Hari

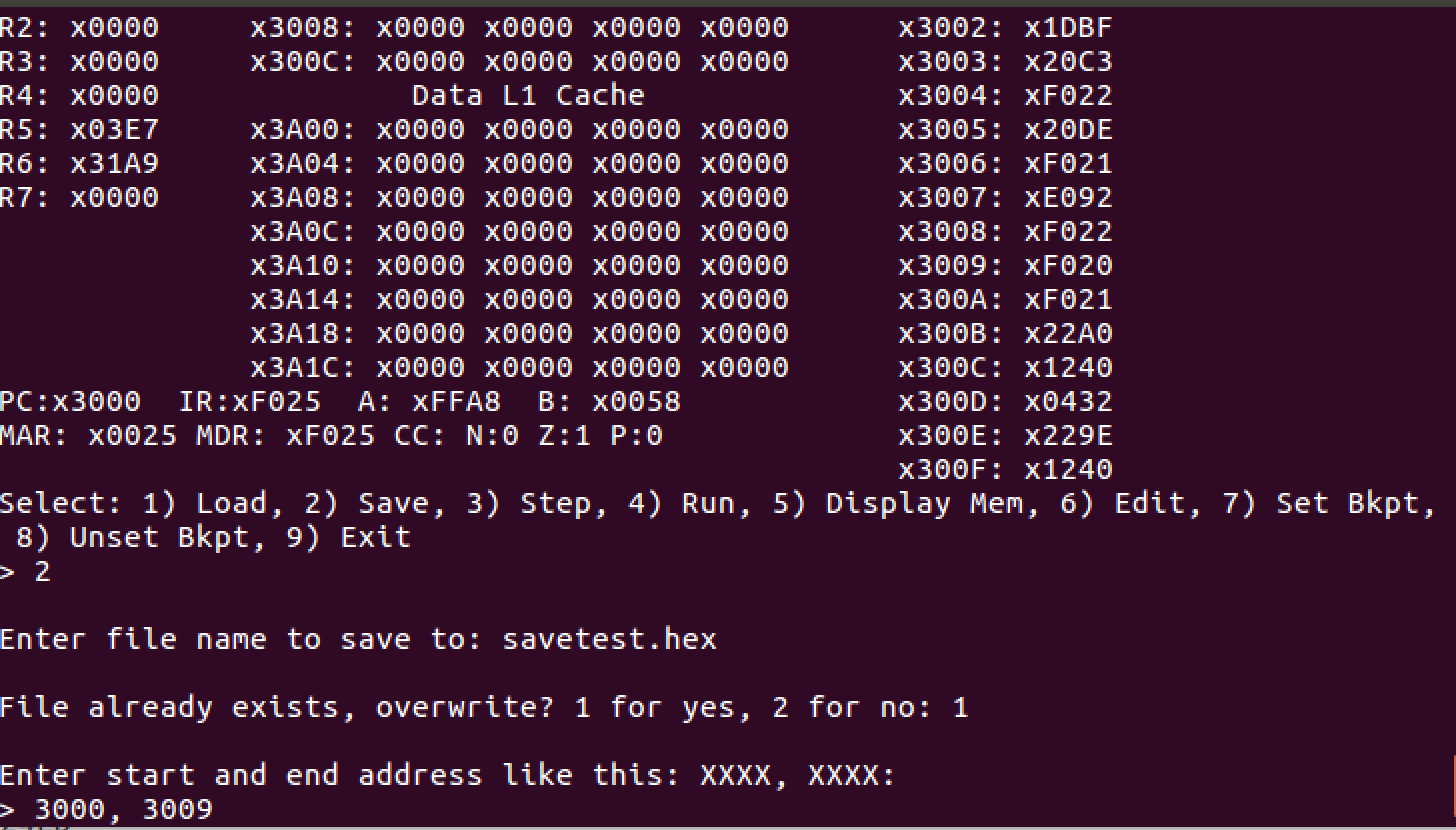
Vladimir Kaganyuk

Dongsheng Han

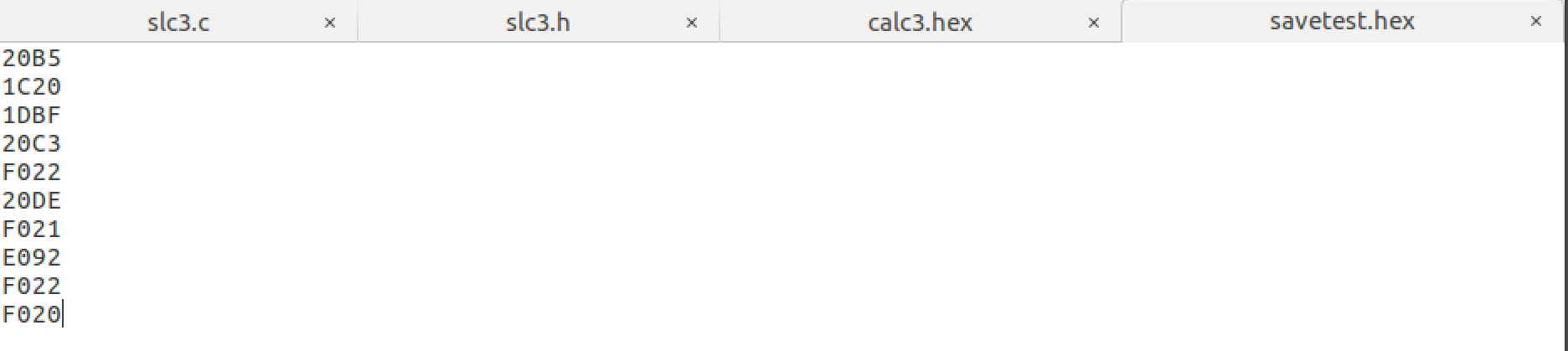
Note: the MAR in our first couple screenshots is incorrect, but you can see in our latest screenshots (and by running our code) that we resolved this issue.

**Save function**

Prompts user for a save file, if it exists a new prompt asks the user if they would like to overwrite that file.

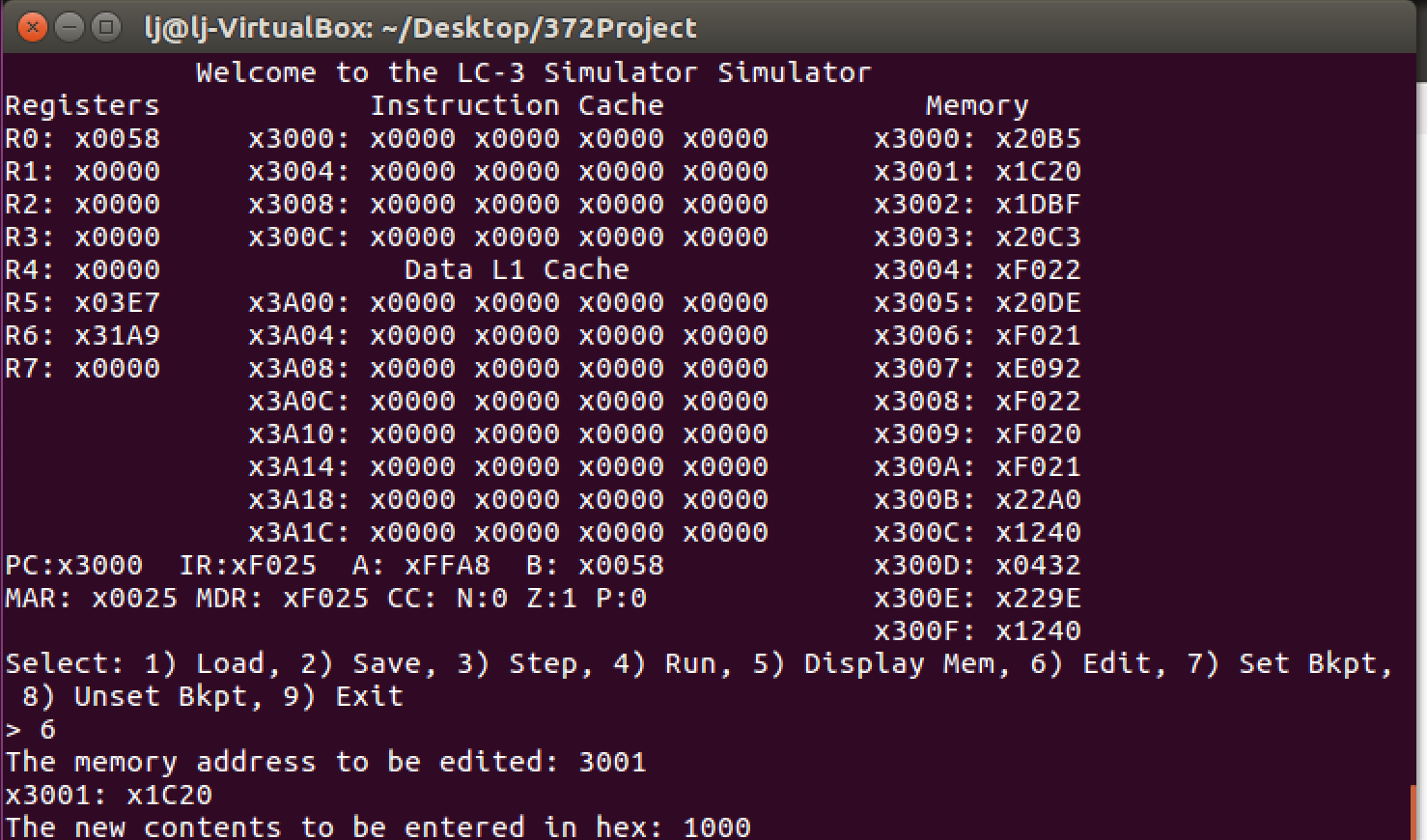


Contents of the save.hex file after displaying from x3000 to x3009

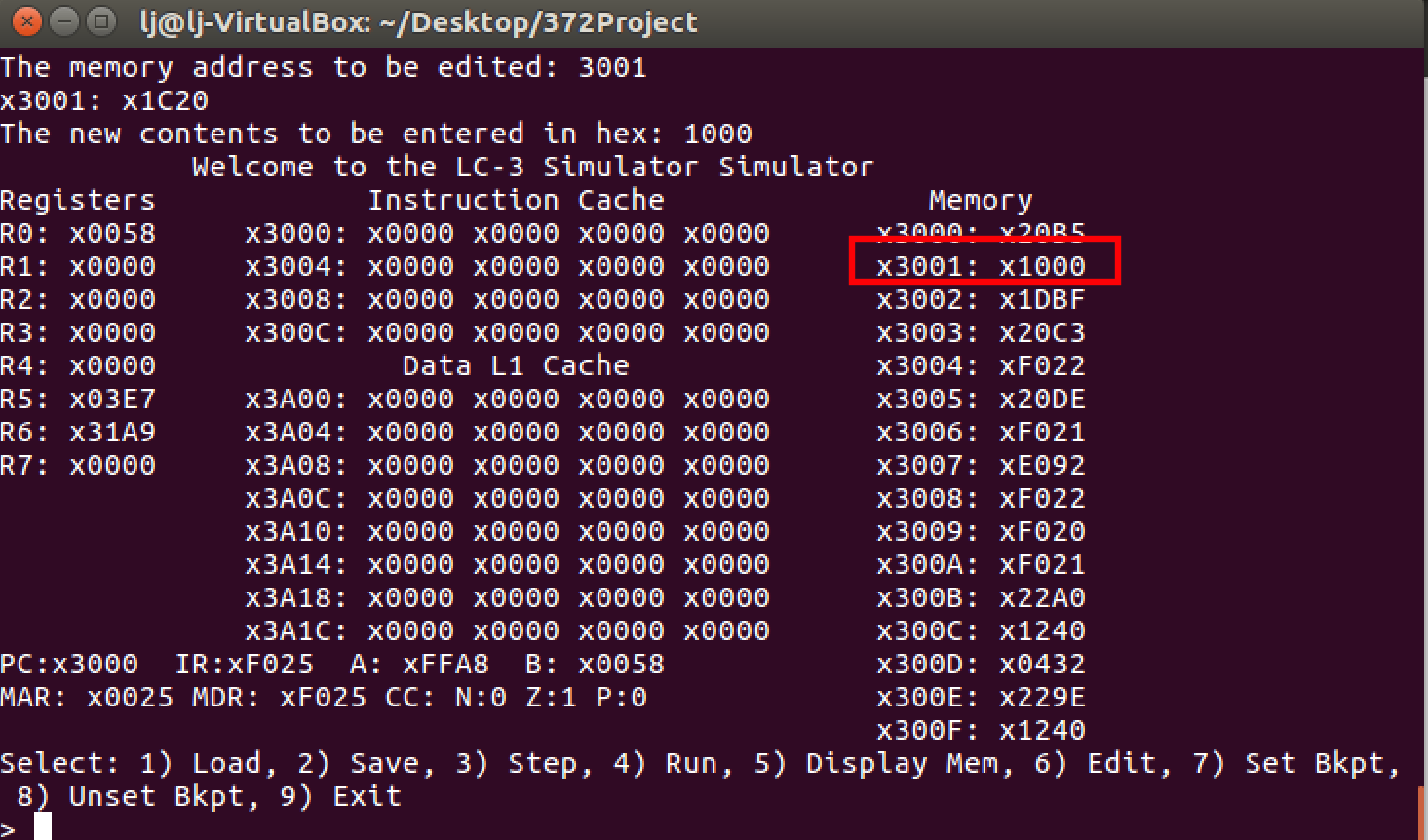


**Edit Function**

The user is prompted for an address to be edited. The address is then displayed showing the contents as well. The user then changes the value at that location.

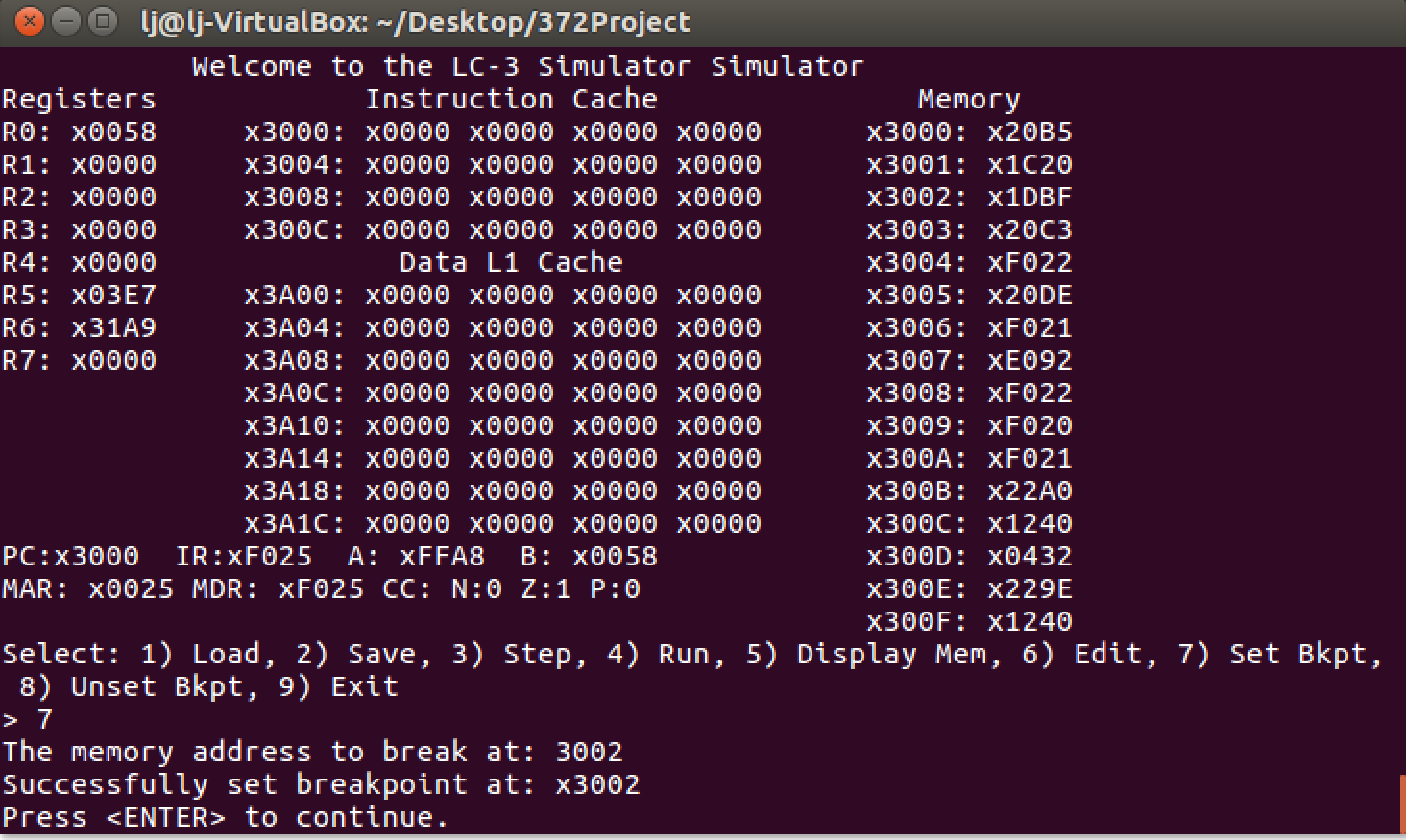


The value entered then changes the value at 3001 while everything else remains the same.

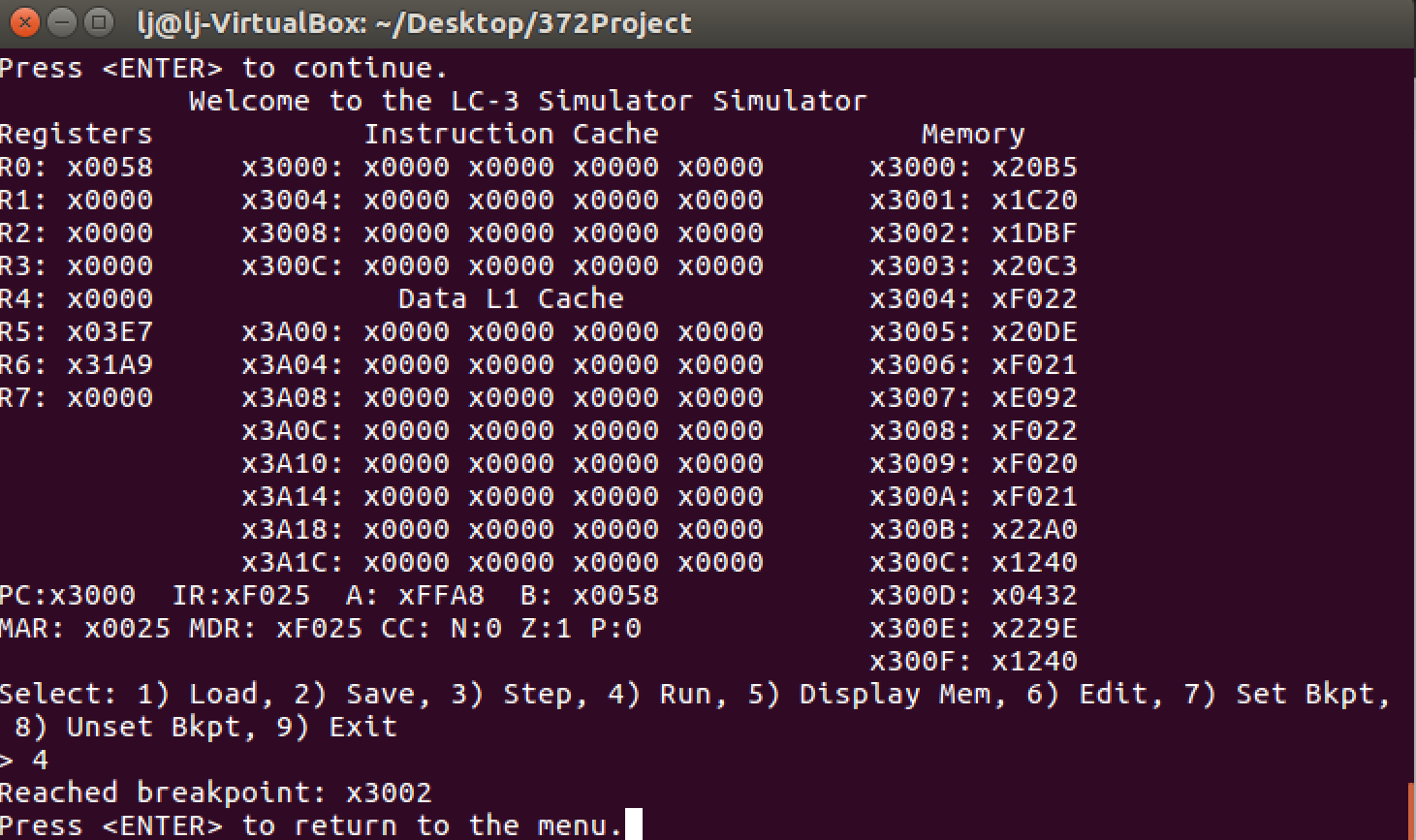


**Set/Unset Breakpoints functionality**

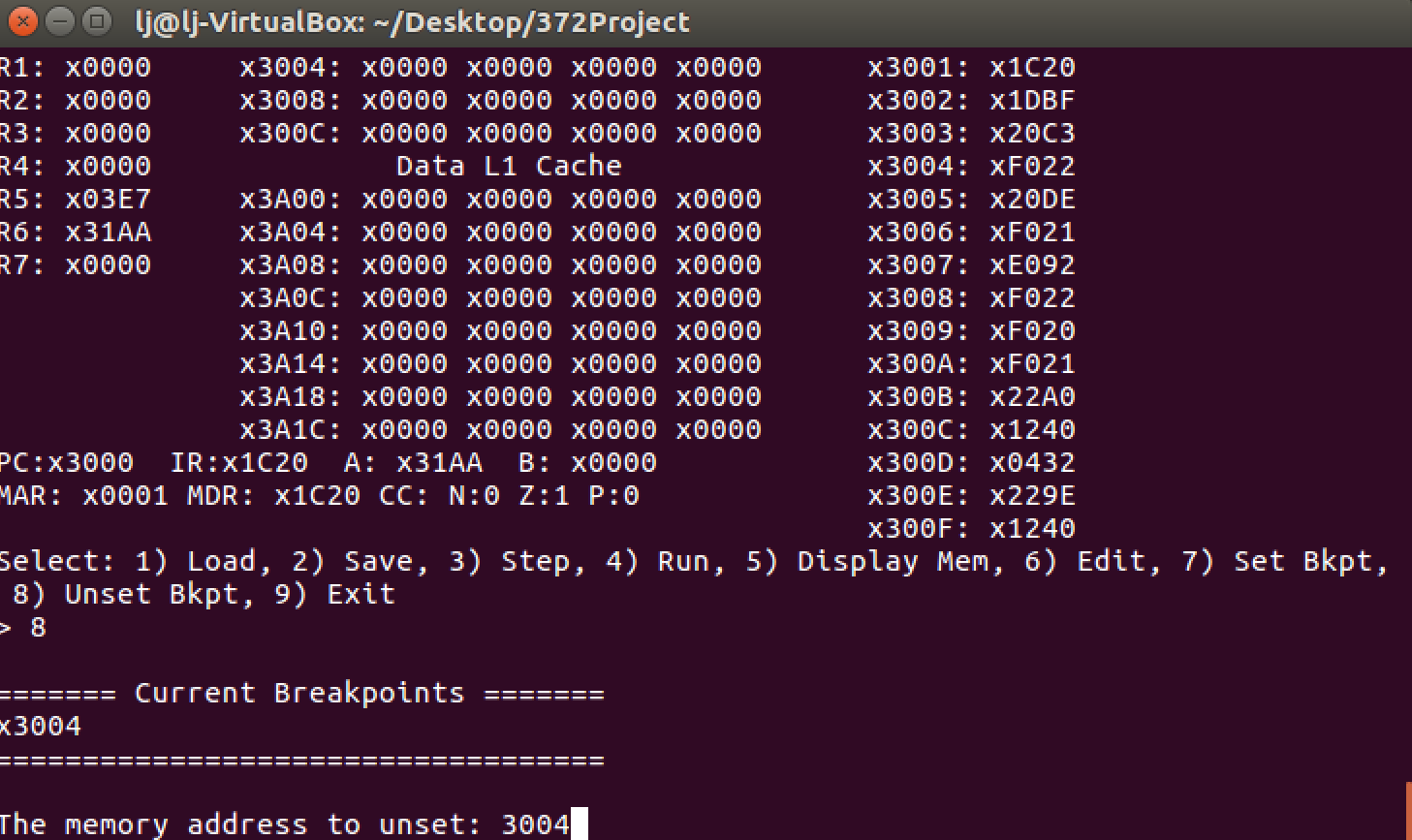
The user is prompted to choose a breakpoint. In the following example I have chosen address 3002 for simplicity purposes.



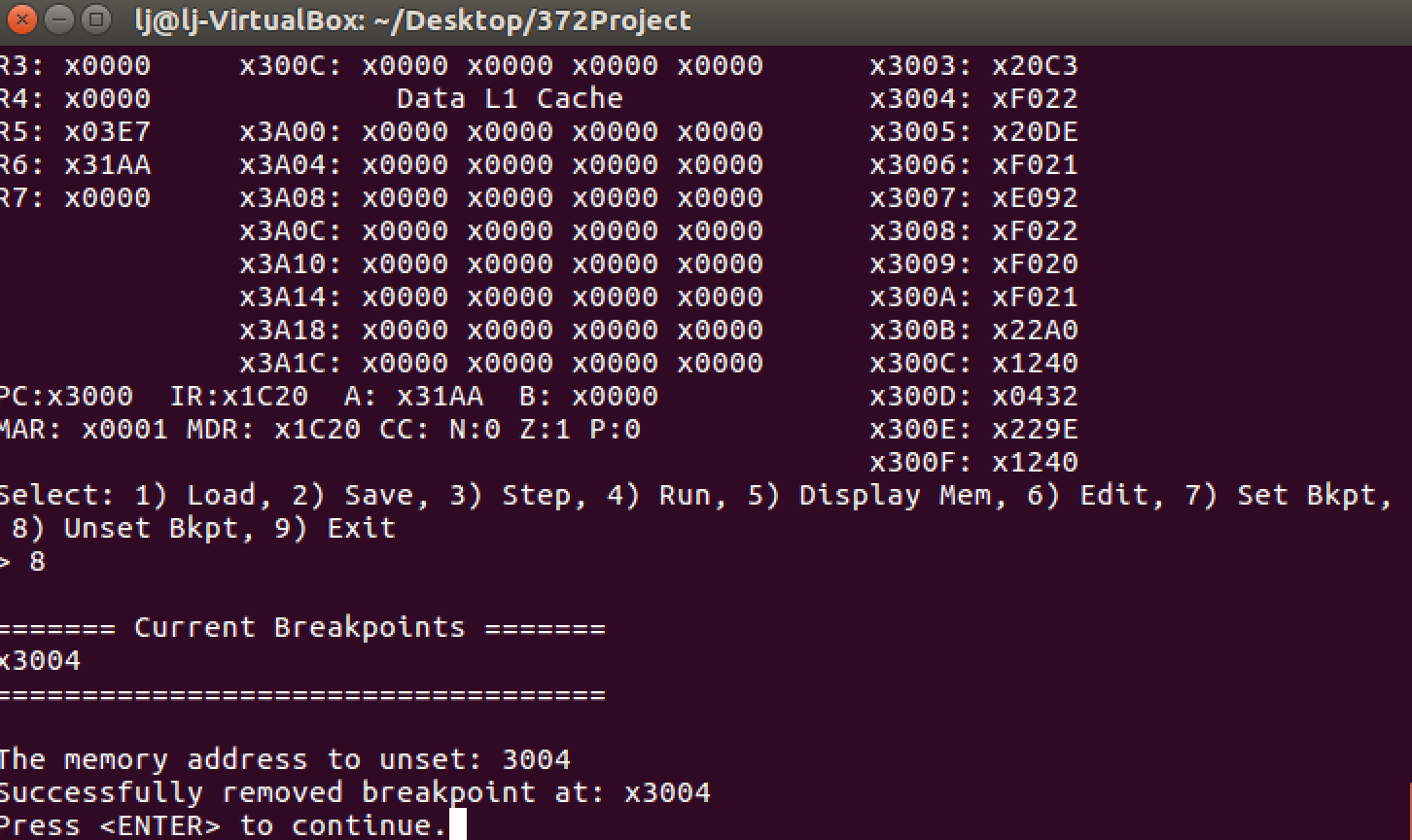
After running the program, the user then gets prompted with a message letting them know they have encountered a breakpoint.



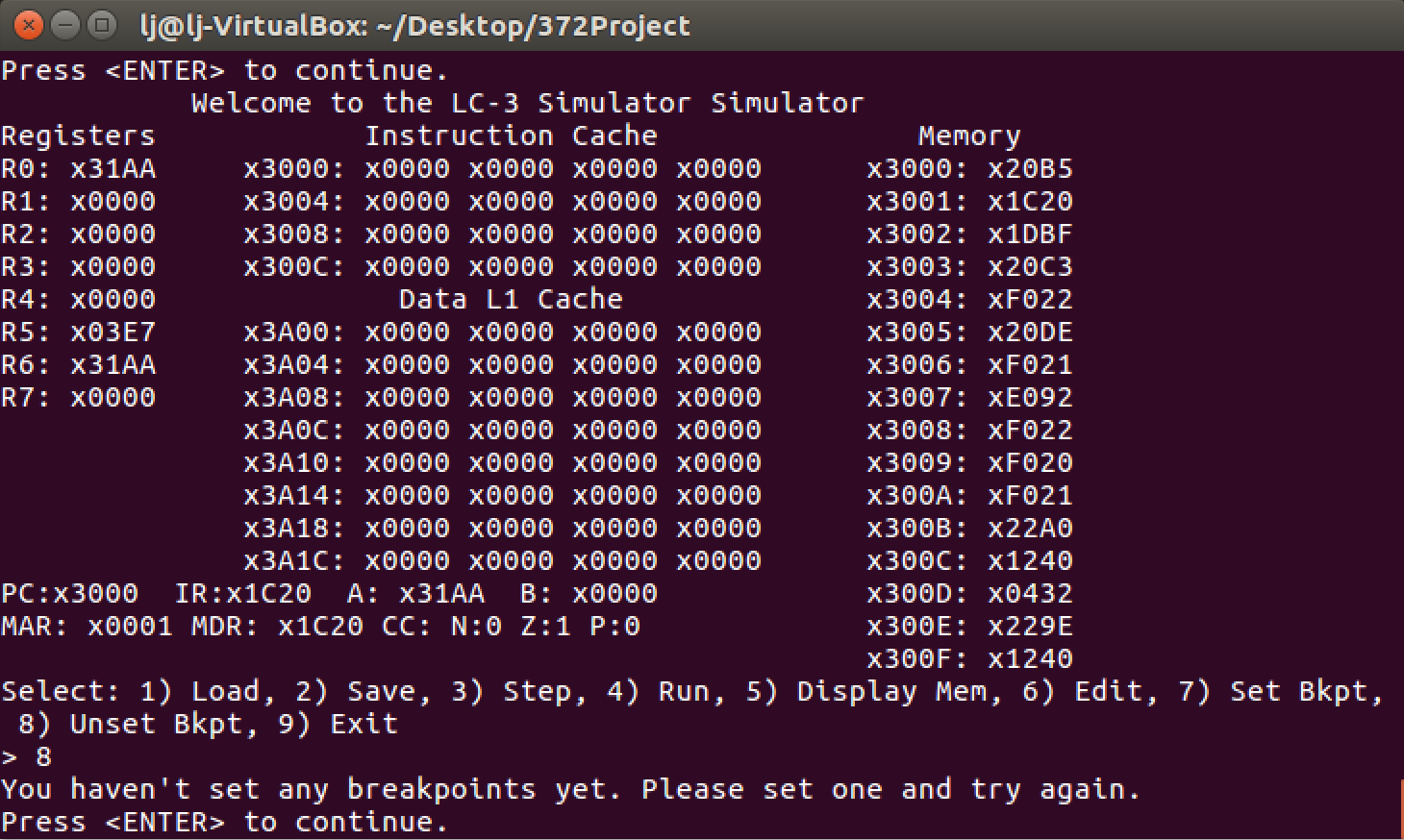
The user then has the option to unset a breakpoint, they are shown what breakpoints are currently set in the program.



The breakpoint is then successfully removed from the program



If no breakpoints exist, then the user is prompted with a message letting them know that they haven’t set any.

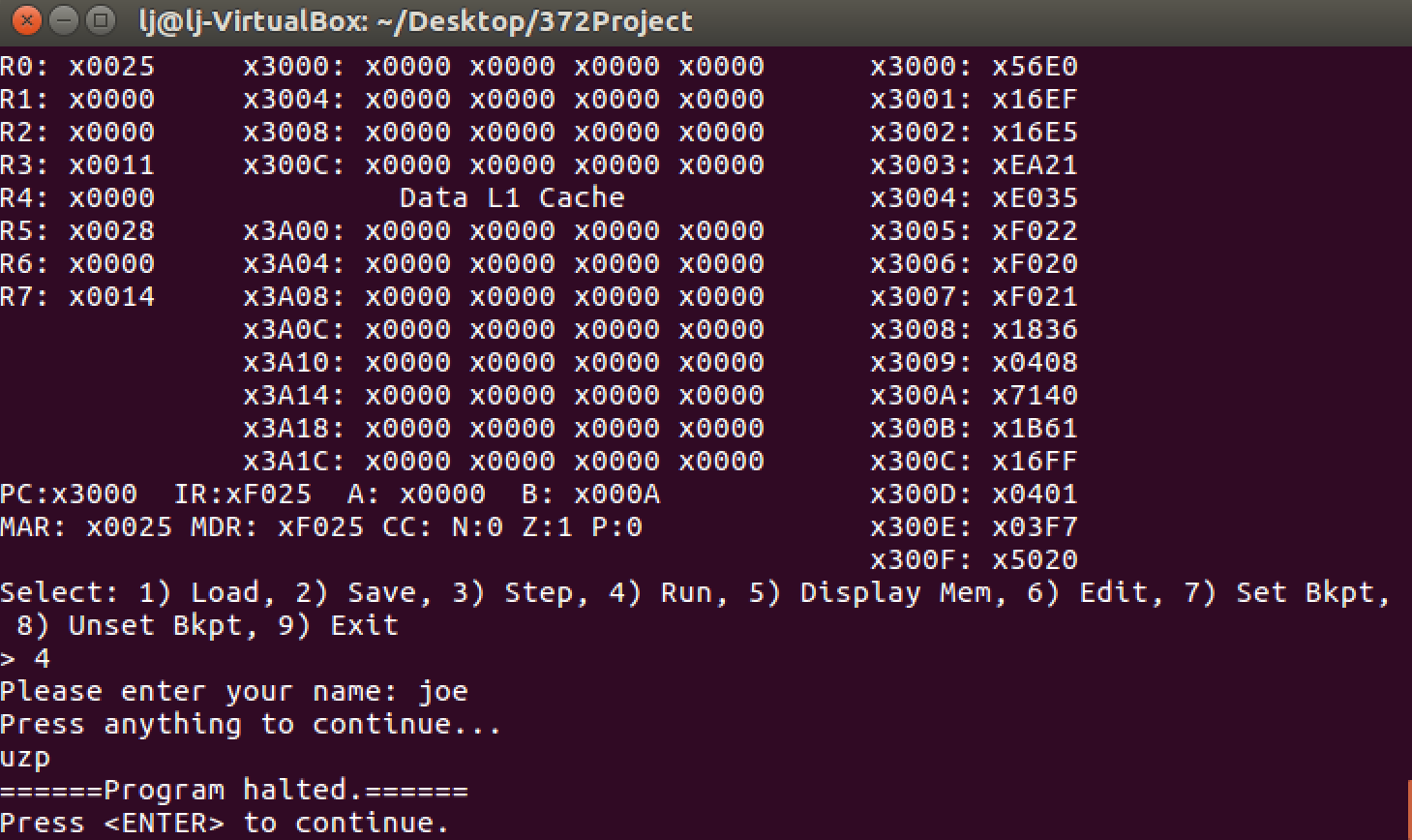


**Cache functionality**

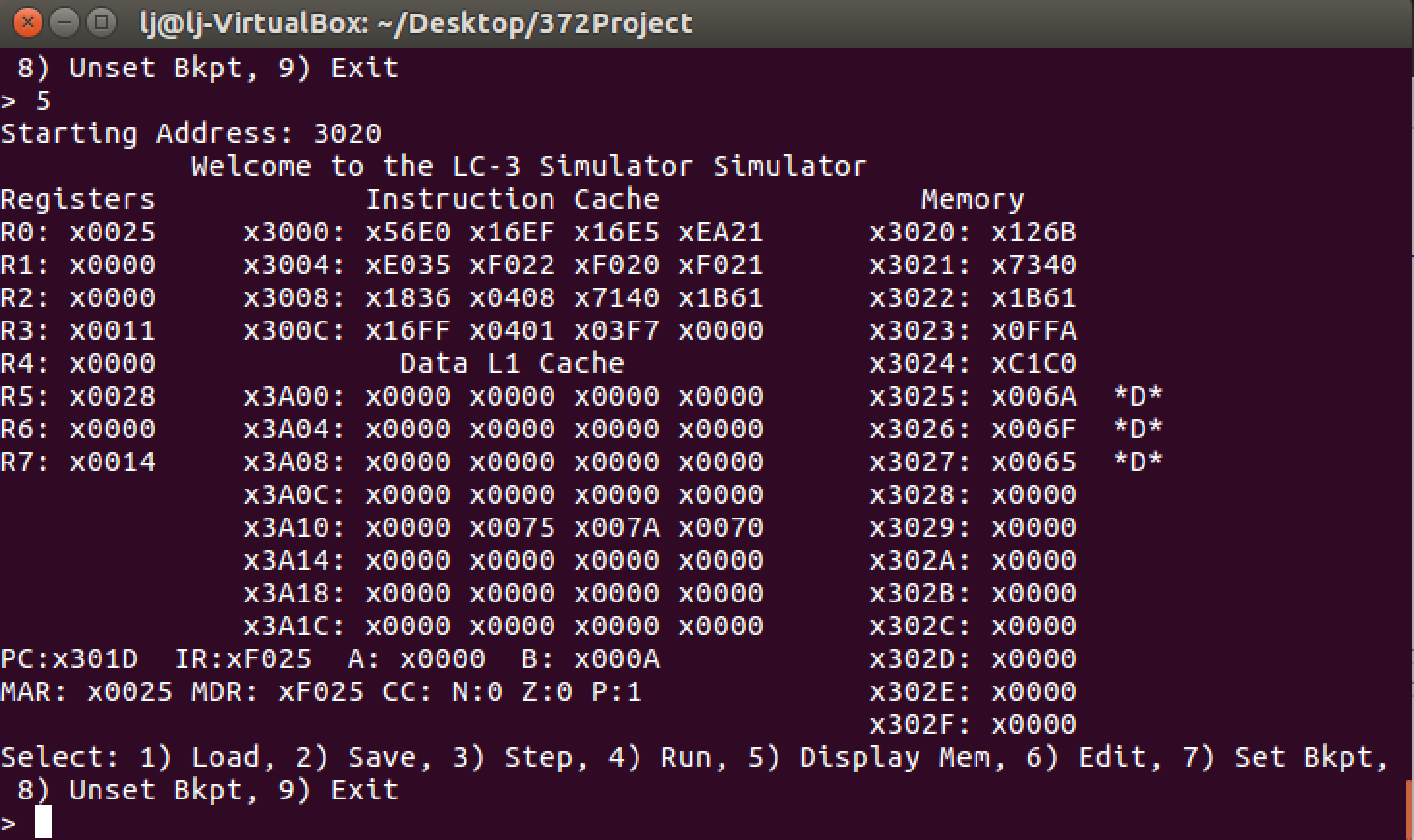
We used a write-back policy for our direct-mapped cache.

Here are the fruits of our labor…

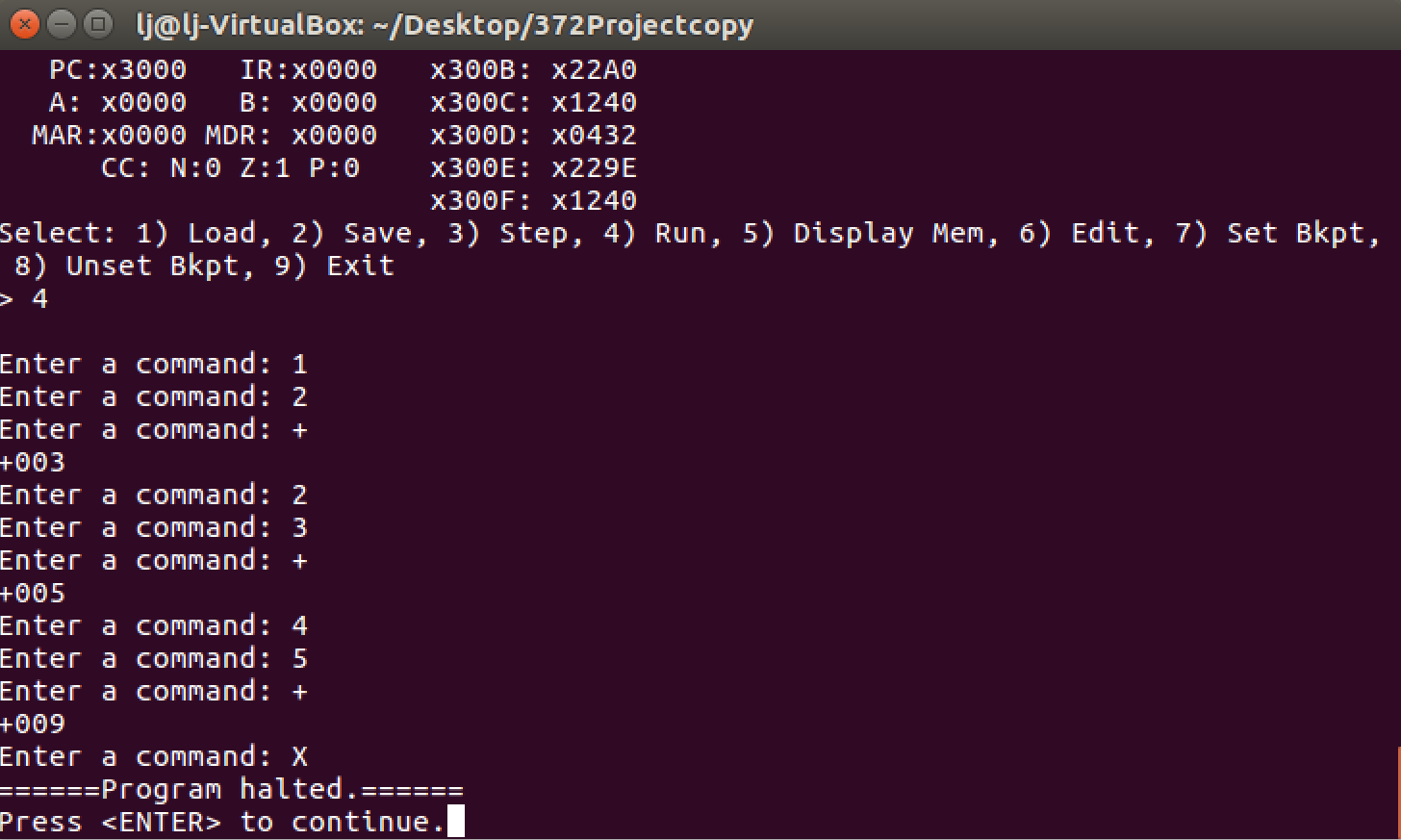
First we enter in a name, for example: joe



Then we display memory starting from location 3020 to show the locations that are inconsistent with the cache (See x3A10 row in Data L1 Cache). These are identified by the \*D\* symbol (D as in dirty).

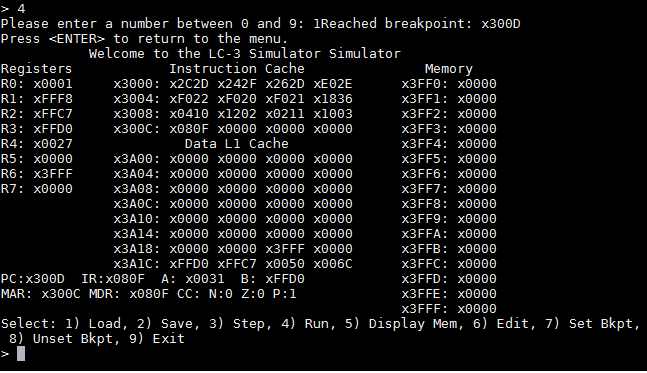


**RPN Calculator**

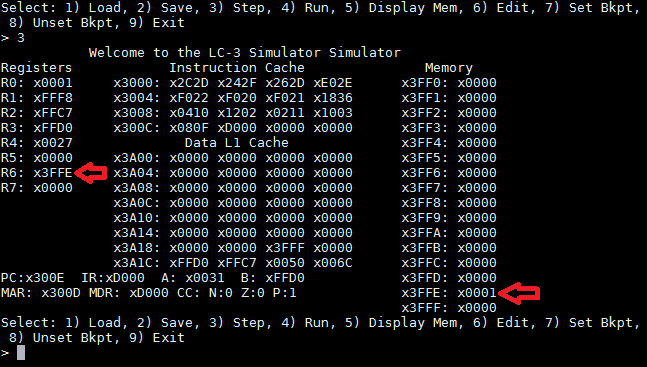


**The Sum Program**

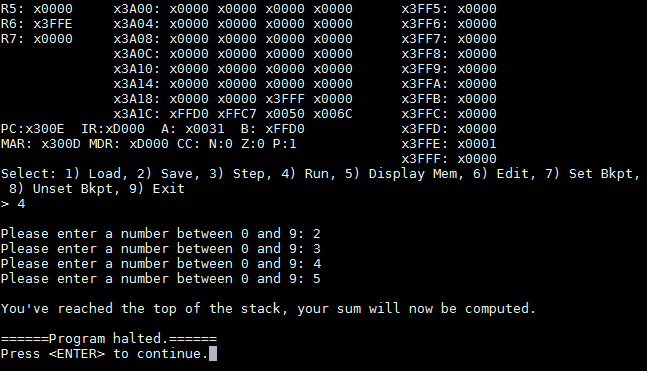
In this first screenshot, I’ve started the program, entered a number (1), and then reached a breakpoint (x300D). This is right before the push functionality.



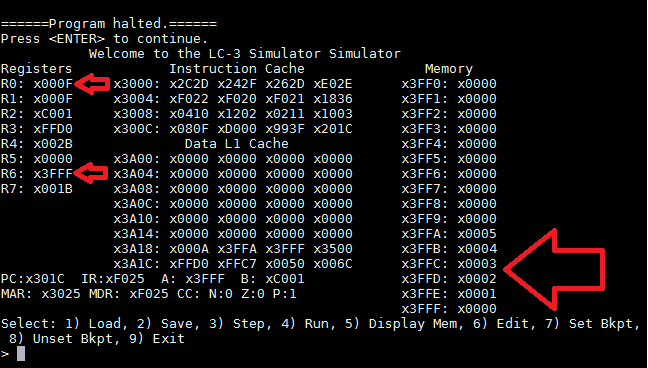
Now I’ve stepped through one instruction (xD000). As you can see, R6 has been decremented, and x3FFE has the value that I typed in (1).



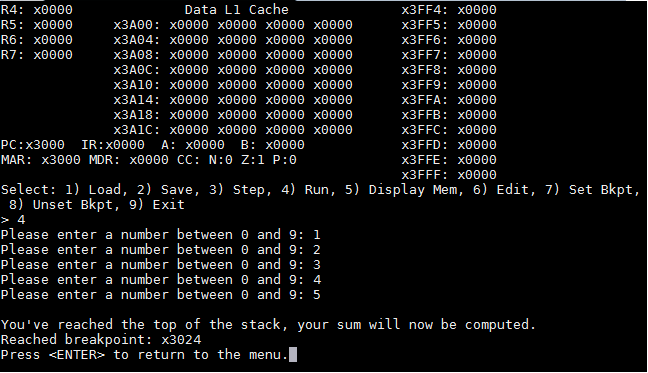
In this screenshot, I’ve gone and typed in four more numbers. As a result, I’ve reached the top of the stack. I’m given a message, and then the program halts.



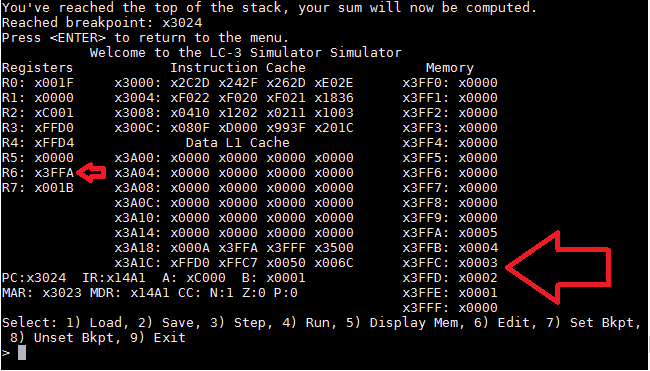
This is the screen that popped up when I pressed Enter. As you can see, R0 now has the sum of the numbers I entered (1, 2, 3, 4, and 5). R6 has returned to the bottom of the stack, x3FFF, but you can still see the remnants of the numbers that were pushed to the stack (x3FFE-x3FFA).



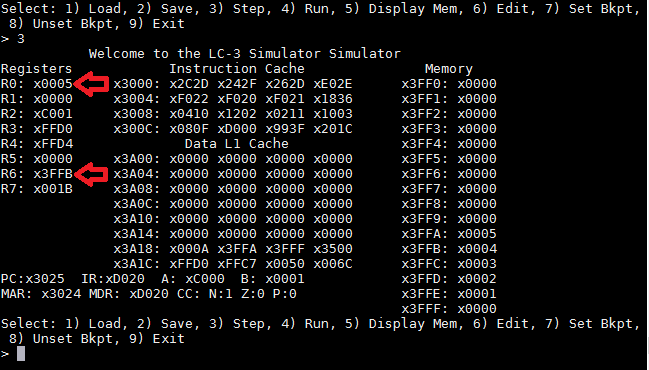
I re-ran the program after setting a breakpoint at x3024 (pop functionality starts).

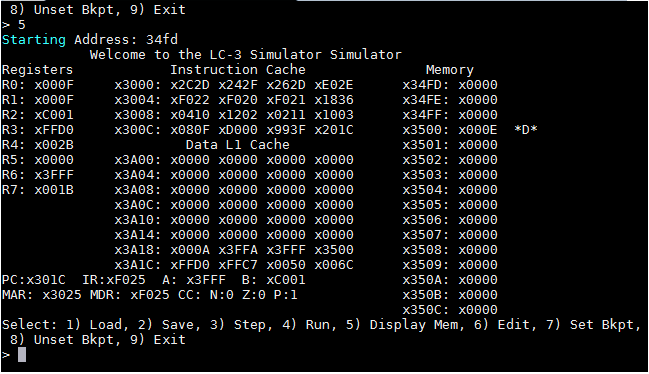


You can see that R6 points to x3FFA, this is because we’ve pushed 5 items onto our stack (the numbers 1-5).



One step into the program, and you can see that the POP instruction (xD020) has executed. The value at x3FFA has been stored into R0, and R6 has been incremented to x3FFB.



Looking at memory location x3500, you can see where the program stored the sum. After every additional number was added, the sum was stored using STI (See Line 47 in addition.asm). It was stored in cache (off-screen), but was written back to memory when the latest sum was stored. In this screenshot, the program has halted, so the cache currently has the correct sum (which was LDI’d into R0), while x3500 has the last sum (x000E), the sum before 1, the digit at the bottom of the stack, was added.