CS 105
Lab 2: Debugger
Playing with X86-64 Assembly

Introduction and Goals

NOTE: This lab must be run on Wilkes. If you run it on a different machine, you may get incorrect answers.

The goals of this assignment are to do some basic investigation of the x86-64 architecture and assembly language, and to begin learning how to use the debugger `gdb`. You can read the manual page for `gdb` by typing "`man gdb`" on the command line.[1] You can scroll up and down in the manual page using the space bar or the up/down and page-up/page-down keys, and can exit by pressing the "q" key. The lab Web page also has links to a quick `gdb` summary and to a printable `gdb` reference card; you can also find other information about `gdb` with Google or another favorite search engine.

Optimizing your learning. In this lab, we will frequently tell you to type certain `gdb` commands. **Pay attention** to those commands; don't just blindly copy them from the handout. In each case, we have carefully selected them because they will be useful to you in this lab, in later labs, and in life. Facility with the debugger is a mark of a good computer scientist; you can find and fix bugs far more quickly with the debugger's help than by inserting print statements or staring at the code.[2]

Compiler options. It will be useful to know that you can get the compiler to generate the assembler source for a program `foo` by running "`gcc -S foo.c`". You should also know that to use the debugger effectively, you will need to compile with the "`-g`" switch. In fact, you should just get in the habit of always compiling with "`-g`"; the situations where it's undesirable are extremely unusual. (But note that `-S` and `-g` are best kept separate.) Also, it's usually wise to compile with the "`-Og`" switch so that optimization doesn't make debugging more difficult. Remember that debugging is nearly always more important than optimization! (Important detail: you can compile with `-S` and any optimization level, and in fact doing so can produce useful insights.)

The manual page for `gcc` is quite lengthy; however you may find it interesting to skim the sections that describe the debug option (`-g`) and optimization options (variants of `-O`). Note that you can search inside a manual page after opening it by hitting the "`/`" key, typing your search, and then pressing Enter. You can use search to quickly locate the sections "Options for Debugging Your Program" and "Options That Control Optimization".

Getting set up. Download the files `problem1.c` and `problem2.c` from the lab Web page. You won't be editing these files, but you will be using them with the debugger throughout the lab.

[1] In general, you can read the manual page for a command (or built-in library function) called `foo` using "`man foo`"

[2] Sometimes those other techniques are useful too, but debuggers are designed to help you find bugs, and they do a good job of it!

Collect your answers to all of the following questions in a plain-text file named “`lab02.txt`”. Identify each section by problem number, and each answer by question number. Be sure to put your name and your partner’s name at the top of the file.

Submission. Submit ONLY the `lab02.txt` answer file, using the following command:

```
cs105submit -a 02 lab02.txt.
```

NOTE: Do not change either of the programs in this lab!

Problem 1—Debugging Optimized Code (17 Points)

Let’s first look at `problem1.c`. You can quickly view the contents of a file using the `less` command, e.g., “`less problem1.c`”. You can navigate through a file using `less` the same way you navigate through `man` pages. (In fact, `man` uses `less` to display the manual!).

This file contains a function that has a small `while` loop, and a simple `main` that calls it. Briefly study the `loop_while` function to understand how it works (you don’t need to fully decode it; just get a clue about what’s going on).

It will also be useful to know what the `atoi` function does. Type “`man atoi`” in a terminal window to view the manual page for `atoi` and find out. (Side note: the function’s name is pronounced “ay to eye,” as in “ASCII to integer”, not “a toy.”)

Finally, it will be useful to have a slight clue about `printf`. Since `printf` is quite complicated, for now we’ll just say that it prints answers, and `"%d"` means “print in decimal”. We encourage you to read more about `printf` in Kernighan & Ritchie or online (the advantage of reading in K&R is that the description there is less complex; recent versions of `printf` have tons of extensions that aren’t particularly useful in this course).

Compile the program with the `-g` switch and with **no** optimization: “`gcc -g -o problem1 problem1.c`”. Run `gdb problem1` and set a breakpoint in `main` (“`b main`”). (When you “set a breakpoint,” you are telling the debugger that whenever the program reaches that line, you want to freeze it so you can type more debugger commands, such as examining variables. It’s much quicker than using `printf`!)

Run the program by typing “`r`” or “`run`”. The program will stop in `main`. (Ignore any warnings; they’re meaningful but we’ll work around them.)

Again, *pay attention* to the commands below. For example lazy (i.e., good) programmers should remember that “`r`” is the quickest way to run a program under `gdb` (see above) and that if you use “`run`” alone it remembers the arguments you used last time (see Step 6 below).

(**Note:** to help you keep track of what you’re supposed to doing, we have used italics to list the breakpoints you should have already set at the beginning of each step—except when they don’t matter. Also, when possible we have listed the state you should be in.)

1. *Existing breakpoint at `main`.*
   Type “`c`” (or “`continue`”) to continue past the breakpoint. What happens?

2. *Existing breakpoint at `main`; after the program terminates.*
   Type “`bt`” (or “`backtrace`”). That will print a “trace” of which function called which to get to where the program died. Take note of the numbers in the left column; they identify the *stack frames* of the calls that led to the point of failure. The point of failure is #0, and `main` is the last function listed. Type “`frame` *n*”, where where *n* is one of those numbers, to get to `main`’s stack frame so that you can look at `main`’s variables. What file and line number are you on?

3. *Existing breakpoint at `main`; after the program terminates.*
   Usually when bad things happen in the library (here, several variants of `strtol`) it’s your fault, not the library’s. In this case, the problem is that `main` passed a bad argument to `atoi`. There are two ways to find out what the bad argument is: look at `atoi`’s stack frame, or print the argument. Figure

out how to look at `atoi`'s stack frame (note that `gcc` sometimes mangles function names a bit). Can you see what argument (`nptr`) was passed to `atoi`? (Answer yes or no.)

4. *Existing breakpoint at `main`; after the program terminates.*
   The lack of information is sometimes caused by compiler optimizations, other times by minor debugger issues. In either case it's a nuisance. Rerun the program by typing "`r`" (you'll have to confirm that you really want to do that) and let it stop at the breakpoint. Note that in Step 2, `atoi` was called with the argument "`argv[1]`". You can find out the value that was passed to `atoi` with the command "`print argv[1]`". What is printed?

5. *Existing breakpoint at `main`; after rerunning the program and stopping at the breakpoint.*
   If you took CS 70, you will recognize that number as the value of a NULL pointer. Like many library functions, `atoi` doesn't like NULL pointers. Rerun the program with an argument of 5 by typing "`r 5`". When it reaches the breakpoint, continue (type "`c`"). What does the program print?

6. *Existing breakpoint at `main`; after the program terminates.*
   Without restarting `gdb`, type "`r`" (without any further parameters) to run the program yet again. (If you restarted `gdb`, you must first repeat Step 5.) When you get to the breakpoint, examine the variables `argc` and `argv` by using the `print` command. For example, type "`print argv[0]`." Also try "`print argv[0]@argc`", which is `gdb`'s notation for saying "print elements of the `argv` array starting at element 0 and continuing for `argc` elements." What is the value of `argc`? What are the elements of the `argv` array? Where did they come from, given that you didn't add anything to the `run` command?

7. *Existing breakpoint at `main`; at `main`.*
   The `step` or `s` command is a useful way to follow a program's execution one line at a time. Type "`s`". Where do you wind up? If you end up on a line inside `atoi.c`, type "`finish`" to get out of `atoi` and then type "`s`" again so that you end at a line inside `problem1.c`. Now answer where you wind up.

8. *Existing breakpoint at `main`; at `main`.*
   `Gdb` always shows you the line that is about to be executed. Sometimes it's useful to see some context. Type "`list`" and the Enter (return) key. What lines do you see? Then hit the Enter key again. What do you see now?

9. *Existing breakpoint at `main`; at `main` and stepped once as described in Step 7.*
   Type "`s`" (and Enter) to step to the next line. Then hit the Enter key three more times. What do you think the Enter key does?

10. *Existing breakpoint at `main`; after stepping once as described in #7 and then stepping four more times.*
    What are the values of `result`, `a`, and `b`?

11. *Existing breakpoint at `main`; after stepping once as described in #7 and then stepping four more times.*
    Type "`quit`" to exit `gdb`. (You'll have to tell it to kill the "inferior process", which is the program you are debugging. Insulting!) Recompile the program, this time optimizing it a bit by adding `-O1` after `-g`: "`gcc -g -O1 -o problem1 problem1.c`". Note that `O` is the letter, not a zero. (Also note that the lowercase "`-o`" is still necessary!) Debug it, set a breakpoint at `loop_while` (not `main`!), and run it with an argument of 10 (not 5!). What line number do you end up at? Remind yourself what `loop_while` contains by typing `list`. At what line number does `loop_while` begin? How strange! (`Gdb` isn't always intuitive with how it matches source code line numbers to breakpoints.)

12. Quit `gdb` again and recompile with `-O2`. Debug the program and set a breakpoint in `loop_while`. Run it with an argument of 20. Where does the program stop?

13. *Existing breakpoint at `loop_while`; after running with argument of 20.*
    Hmmm...that's kind of odd. Disassemble the `main` function by typing "`disassem main`" (or "`disas main`"). The call to `atoi` has disappeared! What has replaced it? (You may wish to refer back to what

you discovered in Steps 2–3.) What is the address of the instruction that calls `printf`? (You will have to do some inference here, because `gcc` mangles the names a bit.)

14. *Existing breakpoint at `loop_while`; after running with argument of 20.*
    That wasn't too hard. But what's up with `loop_while`? Where's the call to it?

15. *Existing breakpoint at `loop_while`; after running with argument of 20.*
    A handy feature of `print` is that you can use it to convert between bases. For example, what happens when you type "`print/x 42`"? How about "`p 0x2f`"?

16. *Existing breakpoint at `loop_while`; after running with argument of 20.*
    We might not have covered it in lecture yet, but functions return results in `%rax`. So the result of `atoi` will be in `%rax` (also known, for this problem, as `%eax`—go figure). After the call to `atoi` (or whatever replaced it) there are a few `lea` instructions (`lea` is `gdb`'s version of `leal` and `leaq`, depending on the destination). The `lea` you want to look at contains a **small** hexadecimal constant. Can you find a matching decimal value (positive or negative) in `problem1.c`?

17. Some of the instructions following the call to `atoi` are of secondary importance, and the rest are pretty hard for novices to decode. But if the value in `%eax` is called a, the important instructions (which are the ones at +32, +35, +37, +40, and +42) calculate $15(a - 16) + 2a - 1680$. (After you've developed a bit more facility with the x86-64 instruction set, it will be worth your time to return to these instructions and analyze them.)

    That's weird. Or not: it turns out that the compiler is so smart that it figured out the underlying math of `loop_while` and replaced it with a straight-line calculation. Wow!

    No answer is needed here.

Problem 2—Stepping and Looking at Data (17 Points)

Now take a look at `problem2.c`. This file contains three `static` constants and three functions. Read the functions and figure out what they do. (If you're new to C, you may need to consult your C book or some online references.) Here are some hints: `argv` is an array containing the strings that were passed to the program on the command line (or from `gdb`'s `run` command); `argc` is the number of arguments that were passed. By convention, `argv[0]` is the name of the program, so `argc` is always at least 1. The `malloc` line allocates a variable-sized array big enough to hold `argc` integers (which is slightly wasteful, since we only store `argc-1` integers there, but what the heck).

By now we hope you've learned that optimization is bad for debugging. So compile the program with `-Og -g` and bring up the debugger on it.

1. `Gdb` provides you lots of ways to look at memory. For example, type "`print puzzle1`" (something you should already be familiar with). What is printed?

2. Gee, that wasn't very useful. Sometimes it's worth trying different ways of exploring things. How about "`p/x` puzzle1"? What does that print? Is it more edifying?

3. You've just looked at puzzle1 in decimal and hex. There's also a way to look at it as a string, although the notation is a bit inconvenient. The "x" (examine) command lets you look at arbitrary memory in a variety of formats and notations. For example, "`x/bx`" examines bytes in hexadecimal. Let's give that a try. Type "`x/4bx &puzzle1`" (the "`&`" symbol means "address of"; it's necessary because the `x` command requires addresses rather than variable names). How does the output you see relate to the result of "`p/x puzzle1`"? (Incidentally, you can look at any arbitrary memory location with `x`, as in "`x/wx 0x404078`".)

4. OK, that was interesting and a bit weird (thanks to all the Little-Endians in Lilliput). But we still don't know what's in `puzzle1`. We need help! And fortunately `gdb` has help built in. So type "`help x`". Then experiment on `puzzle1` with various forms of the `x` command. For example, you might try "`x/16i &puzzle1`". (`x/16i` is one of our favorite `gdb` commands—but since here we suspect that `puzzle1` is data, not instructions, the results might be interesting but probably not correct.) Keep experimenting until you find a sensible value for `puzzle1`. What is the human-friendly value of `puzzle1`?

   Hints:

   - Don't accept an answer that is partially garbage!
   - Although `puzzle1` is declared as an `int`, it's not actually an integer. But on a 32-bit machine an int is 4 bytes, 2 halfwords, or one (in `gdb` terms) word.
   - There are 44 possible combinations of sizes and formats. But you know the size, right? And the "`c`", "`i`", and "`s`" formats don't make sense with a size, so you have a manageable number of choices. Try them all! Be systematic.

5. Having solved `puzzle1`, look at the value carefully. Is it correct? (You might wish to check it online.) If it's wrong, why is it wrong?

6. Now we can move on to `puzzle2`. It pretends to be an *array* of `int`s, but you might suspect that it isn't. Using your newfound skills, figure out what it is. (**Hint:** since there are two `int`s, the entire value occupies 8 bytes. So you'll need to use some of the size options to the `x` command.) What is the human-friendly value? (Hint: it's not "105". Nor is there garbage in it.)

7. Are you surprised?

8. Is it correct?

9. We have one puzzle left. By this point you may have already stumbled across its value. If not, figure it out; it's often the case that in a debugger you need to make sense of apparently random data. What is stored in `puzzle3`?

10. We've done all this without actually running the program. But now it's time to execute! Set a breakpoint in `fix_array`. Run the program with the arguments `1 1 2 3 5 8 13 21 44 65`. When it stops, print `a_size` and verify that it is 10. Did you really need to use a `print` command to find the value of `a_size`? (**Hint:** look carefully at the output produced by `gdb`.)

11. *Existing breakpoint at `fix_array`; stopped at that breakpoint.*
    What is the value of `a`?

12. *Existing breakpoint at `fix_array`; stopped at that breakpoint.*
    Type "`display a`" to tell `gdb` that it should display `a` every time you step (although `gdb` will only obey part of the time). Step five times. Which line last displayed `a`?

13. *Existing breakpoint at `fix_array`; after hitting that breakpoint and then stepping five times.*
    Step twice more (a sixth and seventh time). What is the value of `a` now? What is `i`?

14. *Existing breakpoint at `fix_array`; after hitting that breakpoint and then stepping seven times.*
    At this point you should (again) be at the call to `hmc_pomona_fix`. You already know what that function does, and stepping through it is a bit of a pain. The authors of debuggers are aware of that fact, and they always provide two ways to step line-by-line through a program. The one we've been using (`step`) is traditionally referred to as "step into"—if you are at the point of a function call, you move stepwise *into* the function being called. The alternative is "step over"—if you are at a normal line it operates just like `step`, but if you are at a function call it does the whole function just as if it

were a single line. Let's try that now. In `gdb`, it's called `next` or just `n`. Type "`n`" twice. What line do we wind up at? What is the value of `i` now? (Recall that in `gdb` as in most debuggers, the line shown is the *next* line to be executed.)

15. *Existing breakpoint at `fix_array`; after hitting that breakpoint, stepping seven times, and typing `next` twice.*
    It's often useful to be able to follow pointers. `Gdb` is unusually smart in this respect; you can type complicated expressions like `p *a.b->c[i].d->e`. Here, we have kind of lost track of `a`, and we just want to know what it's pointing at. Type "`p *a`". What do you get?

16. *Existing breakpoint at `fix_array`; after hitting that breakpoint, stepping seven times, and typing `next` twice.*
    Often when debugging, you know that you don't care about what happens in the next three or six lines. You could type "`s`" or "`n`" that many times, but we're computer scientists, and CS types sneer at doing work that computers could do for them—especially mentally taxing tasks like counting to twelve. So on a guess, type "`next 12`". What is the value of `*a` now?

17. *Existing breakpoint at `fix_array`; after hitting that breakpoint, stepping seven times, and (in effect) typing `next` 14 times (whew!).*
    Let's use `n` to verify that it works just like `s` when you're not at a function call. Type `n` until you see a line from `main`. Then type `n` one more time. Which two lines of `main` were displayed?

Finally, a small side comment: if you've set up a lot of `display` commands and want to get rid of some of them, investigate `info display` and `help undisplay`.

Problem 3—Assembly-Level Debugging (18 Points)

So far, we've mostly been taking advantage of the fact that `gdb` understands your program at the source level: it knows about strings, source lines, call chains, and even complicated C++ data structures. But sometimes it's necessary to get down and dirty with the assembly code.

Note: If you get to this point before we've done the lecture on "flow control", this would be a good time to take a break and work on some other class.

Note: When you are working with assembly code, it can be *very* helpful to issue the `gdb` command "`set disassemble-next-line on`". That will tell `gdb` that whenever the program stops, it should disassemble and display the next instruction that is to be executed. We suggest that you issue this command whenever you start gdb.

To be sure we're all on the same page, let's quit `gdb` and bring it up on `problem2` again, still using `-Og -g`. Run the program with arguments of `1 42 2 47 3`.

1. *No breakpoints; after running `problem2`.*
   What is the output? Whoop-dee-doo.

2. *No breakpoints.*
   Set a breakpoint in `main`. Run the program again (use "`r`" alone so that it gets the same arguments). What line does it stop at?

3. *Existing breakpoint at `main`; after running the program.*
   Boooooooooooring. Type "list" and then Enter to see what's nearby, then type "`b 35`" and "`c`". Where does it stop now?

4. *Existing breakpoints at `main` lines 29 and 35; after running and continuing.*
   Shocking. But since that's the start of the loop, typing "`c`" will take you to the next iteration, right?

5. *Existing breakpoints at `main` lines 29 and 35; after running and continuing twice.*
   Oops. Good thing we can start over by just typing "`r`". Continue past that first breakpoint to the second one, which is what we care about. But why, if we're in the `for` statement, didn't it stop the second time? Type "`info b`" (or "`info breakpoints`" for the terminally verbose). Lots of good stuff there. The important thing is in the "address" column. Take note of the address given for breakpoint 2, and then type "`disassem main`". You'll note that there's a helpful little arrow right at breakpoint 2's address, since that's the instruction we're about to execute. Looking back at the corresponding source code, what part of the `for` statement does this assembly code correspond to?

6. *Existing breakpoints at `main` lines 29 and 35; after running and continuing once.*
   The instruction after `main+28` is `main+33`. It does the loop comparison, which jumps to `main+72` if the looping should end. See if you can find the end of the loop body that jumps back to `main+33`. This is all part of the `for` loop pattern we covered in class. We've successfully breaked ("broken?" "set a breakpoint?") at the loop initialization. But we'd like to have a breakpoint *inside* the for loop, so we could stop on every iteration. Note that if no jump occurs after the compare at `main+33`, we enter the loop body at `main+37`. Sometimes we want to break at a given line of assembly, even if it's in the middle of a source line. No worries, though, because `gdb` will let us set a breakpoint on *any* instruction even if it's in the middle of a statement! Just type "`b *(main+37)`" or "`b *0x401194`" (assuming that's the address of `main+37`, as it was when we wrote these instructions). The asterisk tells `gdb` to interpret the rest of the command as an address in memory, as opposed to a line number in the source code. What does "`info b`" tell you about the line number you chose? (Fine, we could have just set a breakpoint at that line. But there are more complicated situations where there isn't a simple line number, so it's still useful to know about the asterisk.)

7. *Existing breakpoints at `main` lines 29 and 35, and instruction `main+37`; after running and continuing twice before hitting the third breakpoint.*
   We can look at the current value of the array by typing "`p array[0]@argc`" or "`p array[0]@6`". But the current value isn't interesting. Let's continue a few times and see what it looks like then. Typing "`c`" over and over is tedious (especially if you need to do it 10,000 times!) so let's continue to breakpoint 3 and then try "`c 4`". What are the full contents of `array`?

8. *Existing breakpoints at `main` lines 29 and 35, and instruction `main+37`; after continuing until breakpoint 3 has been hit and then typing `c 4`.*
   Perhaps we wish we had done "`c 3`" instead of "`c 4`". We can rerun the program, but we really don't need all the breakpoints; we're only working with breakpoint 3. Type "`info b`" to find out what's going on right now. Then use "`d 1`" or "`delete 1`" to completely get rid of breakpoint 1. But maybe breakpoint 2 will be useful in the future, so type "`disable 2`". Use "`info b`" to verify that it's no longer enabled ("Enb"). Run the program again. Where do we stop? (Well, that was hardly a surprise.)

9. *No previous state.*
   Sometimes, instead of stepping through a program line by line, we want to see what the individual instructions do. Of course, instructions manipulate registers. Quit `gdb` and restart it, setting a breakpoint in `fix_array`. (Remember to issue "`set disassemble-next-line on''`.) Run the program with arguments of `1 42 2 47 3`. Type "`info registers`" (or "`info r`" for the lazy) to see all the processor registers in both hex and decimal. Which registers have *not* been covered in class?

10. *Existing breakpoint at `fix_array`; after running and hitting the breakpoint.*
    Well, that's because they're weird and not terribly important. (Except `eflags`, which holds the condition codes—and some other things. Note that instead of being given in decimal, it's given symbolically—things like `CF`, `ZF`, etc.) **Of the flags we have discussed in class,** which ones are set right now? What preceding instruction caused those flags to be set?

    NOTE: If you haven't been through the "x86 control flow" lecture, you will have to return to this step after that.

11. *Existing breakpoint at `fix_array`; after running and hitting the breakpoint.*
    Often, looking at *all* the registers is excessive. Perhaps we only care about one. Type "`p $rdi`". What is the value? Is "`p/x $rdi`" more meaningful?

12. *Existing breakpoint at `fix_array`; after running and hitting the breakpoint.*
    We mentioned a fondness for "`x/16i`".[3] Actually, what we really like is "`x/16i $rip`". Compare that to the result of "`disassem fix_array`". Then, immediately after doing "`x/16i $rip`", hit the Enter key. What do you see? (What function do you see instructions from?)

13. *Existing breakpoint at `fix_array`; after running and hitting the breakpoint.*
    Finally, we mentioned stepping by instructions. That's done with "`stepi`" ("step one instruction"). Type that now, and note that `gdb` gives a new instruction address but says that you're in the left curly brace. If you remembered to do "`set disassemble-next-line on`" then `gdb` will also tell you what instruction you are on.[4] What instruction are we on?

14. Keep hitting Enter to step one instruction at a time until you reach a `callq` instruction. What function is about to be called?

15. *Existing breakpoint at `fix_array`; after hitting the breakpoint and then stepping by instruction until a `callq` is about to be executed.*
    As with source-level debugging, at the assembly level it's often useful to skip over function calls. At this point you have a choice of typing "`stepi`" or "`nexti`". If you type "`stepi`", what do you expect the next instruction to be (hexadecimal address)? What about "`nexti`"? (By now, your debugging skills should be strong enough that you can try one, restart the program, and try the other, so there's little excuse for getting this one wrong!)

16. *Existing breakpoint at `fix_array`; after experimenting with `stepi` and `nexti`.*
    Almost there! Stepping one instruction at a time can be tedious. You can always use "`stepi` *n*" to zip past a bunch of them, but when you're dealing with loops and conditionals it can be hard to decide whether it's going to be 1,042 or 47,093 instructions before you reach the next interesting point in your program. Sure, you could set a breakpoint at the next suspect line. But sometimes the definition of "interesting" is *inside* a line. Let's say, just for the sake of argument, that you are interested in how the `retq` instruction works. You can set a breakpoint there by typing "`b *0x40116e`" (assuming that is its address, as it was when we wrote these instructions). Do so, and then continue. What source line is listed?

17. *Existing breakpoints at `fix_array` and `*0x40117e`; stopped at `retq` instruction.*
    The `retq` instruction manipulates registers in some fashion. Start by looking at what `%rsp` points to. You can find out the address with "`p/x $rsp`" and then use the `x` command, or you could just try "`x/x $rsp`". Or you could get wild and use C-style typecasting: "`p/x *(long *)$rsp`" (try it!). What is the value?

18. *Existing breakpoints at `fix_array` and `*0x40116e`; stopped at `retq` instruction.*
    Use "`info reg`" to find out what all the registers are. Then use "`stepi`" to step past the `retq` instruction, and look at all the registers again. Which registers have changed, and what are their old and new values?

That's it; you're done!

[3] There's nothing special about the number 16; we just like powers of 2, and 16 gives you enough instructions to be useful.

[4] An alternative, which shows only one line, is to use "`display/i $rip`". Don't combine the two techniques or you'll get confused.