Assignment 5: Some assembly required

Due: Mon Nov 13 11:59 pm
Ontime bonus 5%. Grace period for late submissions until Wed Nov 15 11:59 pm

Assignment by Chris Gregg, Julie Zelenski, and Michael Chang

Learning goals

When you have completed this assignment, you will have:

a. written some assembly code that can't be directly produced by C
b. become familiar with the assembly code that `gcc` generates
c. practiced your `gdb` and `gdbtui` skills for investigating assembly code

Overview

This assignment consists of some C and Assembly code study exercises, and some small C and Assembly programs to write. The assembly program you will write will use the "real time clock" that is embedded into all current x86 processors, and will also include working with some floating point assembly instructions.

Get started

Check out the starter project from your cs107 repo using the command

```
git clone /afs/ir/class/cs107/repos/assign5/$USER assign5
```

1. Code study: division

The `division.c` file defines various functions that involve div and mod operations. Compile the program and use `objdump –d` (from shell) or `disassemble` (within gdb) to view the disassembly and answer the following questions in your readme.txt.

a. The x86 integer divide instruction are a bit of an oddball among the arithmetics. Read Section 3.5.5 of B&O to be introduced to these instructions and then review the assembly for the `divide` function to see the instructions in use. The C code shows both a divide and a mod operation, yet the generated assembly has only a single `idiv` instruction. How then is modulus computed?
b. A universal complaint of third graders is that doing long division is tedious; modern CPUs concur. The hard work of division is something the compiler will try to employ wizardry to avoid. Examine the generated assembly for the `udiv_by_two` function. What operation is used to implement unsigned division by a constant 2? Change the function to divide by 32 instead and recompile. What changes in the generated assembly?
c. Whereas add/sub/mul operate equivalently on signed and unsigned integers, it's not so with division. `div_by_two` wants to use the same trick as `udiv_by_two` but needs a little extra tweaking to make it work. Carefully trace the assembly for `div_by_two` to understand how/why it operates. In your readme explain what is different about signed division from unsigned and what the assembly is doing to compensate for that difference.
d. Change the `div_by_two` function to divide by 32 instead and recompile. Its assembly will now contain an instruction you have not yet seen: `cmovns` . Do a web search to learn about this instruction and explain how it is used here.

Further explorations for the curious: Division by powers of two get special treatment, sure, but what about constants that are not powers of two? Look at the C source and generated assembly for `udiv_by_ten` . The assembly don't include any div instruction, by instead there is a multiply by a bizarre number: `0xcccccccd` . What wickedness is this? This mysterious transformation is effectively multiplying by 1/10 as a substitute for divide by 10. The 1/10 value is being represented as a "fixed point fraction" which is constructed similarly to the way floats are. Enter `0.1` into the float visualizer tool (https://www.h-schmidt.net/FloatConverter/IEEE754.html) and look to see where that same funny `0xcccccccd` number shows up in the significand bits of the float. This technique is known as *reciprocal multiplication* and gcc will generally convert division by a constant in this way. The math behind this is somewhat complex and we'll won't make a PhD thesis out of it here, but if you'd like to know more, check out Doug Jones' tutorial on reciprocal multiplication (http://homepage.cs.uiowa.edu/~jones/bcd/divide.html).

2. Write Assembly Code: timer.s

On the x86 processor, some instructions are not accessible directly from C, so sometimes we have to drop into assembly language to access them. One example is the instruction to read the "real time-stamp counter" that is embeded inside the processor. The counter gives us a 64-bit value that represents the integer number of processor "clock ticks" that have occurred since the computer was booted (or restarted). The counter can provide extremely precise timing information for our programs, down to the nanosecond resolution. Here are the details about the machine instruction:

```
Instruction                        Description
   rdtsc      Loads the real time-stamp counter value into %edx:%eax
```

As indicated, the instruction populates the `%edx` and `%eax` registers, and for historical reasons the 64-bit value is shared between these two registers.

You are to write an assembly language function (in the `timer.s` file) that returns a `long` value that represents the number of **milliseconds** since the computer booted:

```
long milliseconds(long clock_frequency_hz);
```

Specific details of the function's operation:

- You will need to use the `rdtsc` instruction to retrieve the number of clock ticks since the computer was rebooted, and convert it to its full 64-bit value.
- The clock frequency passed in is in Hz, and you will have to do some arithmetic to calculate the number of milliseconds based on knowing the number of clock ticks (provided by `rdtsc`) and the frequency.
- You should use the handy one-page x86-64 guide (https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf) to determine which instructions you will need, and to determine which register will hold the parameter, and which register should hold the return value.
- *Note*: If you want to get the processor speed for a myth machine in Hz, you can use the following command:

  ```
  printf "`cat /proc/cpuinfo | grep -i mhz | cut -d: -f2 | head -1 | sed
  's/^ *//'` * 1000000 /1\n" | bc
  ```

Answer the following in your readme.txt file:

a. The `rdtsc` timer counts clock ticks as a 64-bit value, and it can overflow to zero. Based on a clock frequency of 3158610000Hz, how long would it take for the computer to overflow this value?
b. Based on your testing of the time it takes for the exponentially-growing fib() function, what input of `n` to your codetimer program would be likely to be the smallest value that would produce an incorrect result? Assume that you run `./codetimer n 3158610000` within one minute after the computer reboots.

3. Code study: loops

The `loops.c` file defines various `sum` functions to compute the sum of the first N integers. Each function takes a slightly different approach so as to explore each of the C loop constructs `for`, `while`, and `do-while`. (If you have not heard of do-while, first go explore in your C reference). Compile the program and use `objdump -d` or `disassemble` in `gdb` to view the disassembly and answer the following questions in your readme.txt.

a. Read the generated assembly for the `for` and `while` loops. What, if anything, is different about the two?
b. Compare the assembly for those loops to that of the `do-while` loop. What is different about the control flow through a `do-while`? Where can you see that difference in the generated assembly?
c. For each loop type, identify the assembly sequence that corresponds to one iteration of the loop and count those instructions. What percentage of the instructions correspond to "loop control" (i.e. loop increment and test) as opposed to the "interesting" work being done in the loop body?
d. Consider an ordinary for loop such as the one in `sum_A`. On a piece of scratch paper, sketch the sequence of instructions that correspond to a literal top-down rendering of the loop control flow: i.e. loop init first, followed by loop test, loop body, loop increment, and finally a jump back to test. As you may have already noticed, gcc does not sequence the assembly for a loop in this way. The rearrangement instead preferred by the compiler mixes things up and in the process manages to boot one instruction from inside the loop

to outside. What type of instruction is moved? Why is this rearrangement particularly desirable?

e. Look at `sum_D` to see its approach to computing the sum using a technique known as *loop unrolling*. The idea is to do the work of multiple iterations in each loop body and be able to iterate correspondingly fewer times. The overhead of the loop control itself can be a significant factor when the body of the loop itself is quite small; better amortizing that overhead can be a performance boost. Look at the C source for `sum_D` and then examine its assembly. The loop body in `sum_D` does the work of 4 iterations of the loop in `sum_A`, yet the loop body's assembly sequence is only one instruction longer. Explain how this is possible.

f. Run the `loops` program and observe the results printed from timing the different version of `sum`. Note that the timer functions we are using are a bit coarse and subject to artifacts, so do a few trials and discard outliers. Relate the timing results to the assembly being executed for each of the four functions. Are there any unexplained discrepancies between what you expect and what you observe?

Curious about the `repz retq` that shows up in the `sum` functions? Read the assignment faq (advice.html#faq).

4. Reverse-engineer busybox sort

The `samples` subdirectory contains a sorting program called `bbsort` that was compiled from the source published in the busybox project (https://busybox.net).

`bbsort` is a cousin of the `mysort` program you wrote in assign4 (/class/cs107/assign4/) — each implements a simplified version of the standard unix `sort` command. The `bbsort` program is given to you in compiled form only, but even without accessing the C source used to build it, you can discern quite a lot about how it operates. Running the program on various inputs, using `objdump` to review its assembly, and poking around in `gdb` and examining the program state while executing are all techniques that allow you to make observations about the program's inner workings. Working backwards from the compiled assembly to construct a picture of the original C source is a process known as *reverse-engineering*. Let's try this on `bbsort`! Try out the tasks listed below and write up what you learn in your readme.txt.

a. The first question is to figure out what command-line options are supported by `bbsort`. If you invoke standard sort as `sort --help`, it responds with usage for the command, including its giant list of supported flags. Try `bbsort --help` to see its usage. Which flags does `bbsort` support?

b. Use `bbsort` on the `samples/numbers` file with various combos of command-line flags and verify that the program properly implements the flags it claims to support. But what about the other flags from standard sort? If you invoke `bbsort` with those flags, we expect them to be rejected as invalid usage, but that's not quite what is observed. Try some of the standard sort flags on `bbsort`. Find two unsupported flags that get distinct results and report how each is handled.

c. Start gdb on `samples/bbsort` and run on the `samples/numbers` file. You can run any program under the debugger, not just those that you wrote yourself. You won't be able to do things that require the source for `bbsort` (e.g. cannot `list main` for example), but you can run/step, set breakpoints, examine registers and memory, disassemble and more. Set a breakpoint on `qsort` and run the program on the `samples/numbers` file and you will learn that `bbsort` calls the library function (check-plus to busybox for re-use rather than reimplement!). When the breakpoint on `qsort` is hit, look at the qsort arguments, in particular, the comparison function used to order elements. Re-run sort a few times, each time with different flags, and you'll discover that the same comparison function is used for every sort, regardless of sort options. What is the name of this all-purpose comparison function? Disassemble it and count the number of assembly instructions it contains.

d. Set a breakpoint on this comparison function. When that breakpoint is hit, you want to print the two element values being compared. The arguments to the function are not accessible by name, but you can still access their contents. How? The type of the arguments to the comparison function in the C source would have been `const void *`, but what is the actual type of the arguments? Give the gdb expression that can be used to print the value of the first element to the comparison function when stopped at its breakpoint. Hint: you will need a typecast and, as always, be careful with the level of indirection when interpreting a `void*`.

e. The all-purpose comparison function used by `bbsort` uses the command-line flags to determine which kind of comparison to apply to the elements. The function accesses this information via a global variable (boo!). The sort main function calls the getopt32 function to parse the command-line options which records the settings in a global variable which is later read by the comparison function. Read through the disassembly for the comparison function to identify where it reads from this global variable. (as a hint for where to look,

the compiler adds a comment to the instruction that accesses a global variable to identify the symbolic name of the variable). Ask gdb to print the value of the global variable. Run the program with various command-line flags and see how the value changes based on the chosen options. (It may be helpful to print the value in binary or hex to better see what is going on). What is the value of the global variable if the program is invoked with the flags `-n -r`? What if invoked with just `-r`?

f. Set a breakpoint on `qsort` and when hit, print the elements from the array being sorted (review gdb protip from lab4 (/class/cs107/lab4)), use the gdb `finish` command to execute the rest of the `qsort` call, and then print the array again to see how it was sorted. From these observations, determine how `bbsort` handles the `-r` and `-u` flags. For `-r`, is the array actually sorted in reverse or just printed in reverse? For `-u`, are duplicates removed from the array before or after sorting?

Great job! Your sleuthing skills are well on their way to being ready for the famous binary bomb coming up as your next assignment.

Advice/FAQ

Don't miss out on the good stuff in our companion document!

Go to advice/FAQ page (advice.html)

Grading

This assignment is almost entirely readme-based, with the exception of the small amount of assembly code you will write for the timer. We estimate something in the neighborhood of 40 points allocated to the readme answers and 20 points to the timer.

On-time bonus (+5%)

The on-time bonus for this assignment is 5%. Submissions received by the due date earn the on-time bonus. The bonus is calculated as a percentage of the point score earned by the submission.

Finish and submit

Review the How to Submit (/class/cs107/submit.html) page for instructions. Submissions received by the due date receive the on-time bonus. If you miss the due date, late work may be submitted during the grace period without penalty. No submissions will be accepted after the grace period ends, please plan accordingly!

How did it go for you? Review the post-task self-check (/class/cs107/selfcheck.html#assign5).