CMSC216: Assembly Review A

Chris Kauffman

Last Updated:
Sun Oct 25 05:12:30 PM EDT 2026

Exercise: Problem 1

Write an assembly version of the nearby C function `scale_big_ans_twice()`

```
1  long big_ans = 0;
2
3  int scale_num(long *num, long scale);
4
5  void scale_big_ans_twice(long factor){
6    if(big_ans == 0){
7      return;
8    }
9    scale_num(&big_ans, factor);
10   scale_num(&big_ans, factor);
11   return;
12 }
```

Answer: Problem 1

```
1  .data
2  .global big_ans
3  big_ans:
4          .long 0
5
6  .text
7  .global scale_big_ans_twice
8  scale_big_ans_twice:
9          pushq   %rbx
10         cmpq    $0,big_ans(%rip)
11         je     .DONE
12         movq    %rdi, %rbx
13         leaq    big_ans(%rip), %rdi
14         movq    %rbx, %rsi
15         call    scale_num
16         leaq    big_ans(%rip), %rdi
17         movq    %rbx, %rsi
18         call    scale_num
19 .DONE:
20         popq    %rbx
21         ret
```

Discussion Questions

1. How is the global variable `big_ans` used in assembly? What lines use it and what are they doing?
2. Is the stack aligned for the function call `scale_num`? Describe how that alignment looks and the instructions achieve it?
3. Why is the `rbx` register used here? Why is it pushed / popped? Why are no other registers push / popped?

Answer: Problem 1 Discussion Questions Answers 1 / 2

1. How is the global variable `big_ans` used in assembly? What lines use it and what are they doing?
   ANSWER: `big_ans` is accessed early in the instruction
   `cmpq $0,big_ans(%rip)`
   which compares it to 0. `big_ans` appears later in the instructions
   `leaq big_ans(%rip), %rdi`
   which load the address of the global variable as arguments for function calls.

2. Is the stack aligned for the function call `scale_num`? Describe how that alignment looks and the instructions achieve it?
   ANSWER: On entering the function, the stack is 8 bytes away from being divisible by 16 bytes as a return address will have been pushed onto the stack when it is called. The `pushq` instruction at the start of the function extends the stack by 8 bytes to save the old value of `rbx`. This aligns the stack properly for the function call to `scale_num`.

Answer: Problem 1 Discussion Questions Answers 2 / 2

3. Why is the `rbx` register used here? Why is it pushed / popped? Why are no other registers push / popped?
   ANSWER: On entering the function, `%rdi` holds the `factor` parameter, an integer that must be passed to `scale_num()`. Since `%rdi` is a caller save registers, it may change across the function calls and `%rdi` needs a different parameter for the call to `scale_num()`. To retain the value of `factor` across the funciton call, it is copied to `%rbx` which is a caller save register and stable across the function call boundary. However, changes to callee save registers must be undone, thus the push/pop in the function which retains and restores the previous value of `rbx` before returning.

Exercise: Problem 2

Secon Drag is analyzing a binary and attempting to determine what kind of output funciton call `file_print()` will put into an output file. Below is some code which he pulls up in GB for the function.

```
...
0x01158 <file_print+15>: lea    0xea5(%rip),%rdx
0x0115f <file_print+22>: lea    0xed2(%rip),%rsi
0x01166 <file_print+29>: mov    0x2eb3(%rip),%rdi
0x0116d <file_print+36>: mov    $0x0,%eax
0x01172 <file_print+41>: call   0x1040 <fprintf>
0x01177 <file_print+46>: test   %eax,%eax
0x01179 <file_print+48>: js     0x1180 <debug_print+55>
0x0117b <file_print+50>: add    $0x8,%rsp
0x0117f <file_print+54>: ret
...
```

Give some **specific steps and GDB commands** Secon could use in this situation to get insight into what type of data will be placed in the file by the function.

Answers: Problem 2 Answers 1 / 2

- Set a break point at the function via something like `break file_print`, run the program, and then `stepi` (step instructions) forward to the call to `fprintf()`

  Alternatively doing a `break *file_print+41` and running will stop at the call location
- At this point, all the argument registers will be loaded with arguments to `fprintf()` and inspecting them will be helpfule
- The first argument `%rdi` is the file used for output; is not of great consequence but may be helpful
- The second argument `%rsi` should be a Format String like `"%s %d"` or `"%08x %c %-s"` or something similar which will show the type of data to output and how it will be formatted

Answers: Problem 2 Answers 2 / 2

- `%rsi` is loaded a few instructions before the call to `fprintf()` in `lea 0xed2(%rip),%rsi`; it is loaded with a global value using the RIP-relative addressing; this is common for string constants like Format Strings
- The format string can be inspected in GDB after that instruction via something like `x/s $rsi` which will show the specific string and reveal the number of items that will be printed and their data types
- After seeing the format string, additional argument registers like `%rdx / %rcx / etc.` can be inspected to see what specific values they hold. Their interpretation will reflect the Format String such as a `%s` for a pointer to a string or `%08x` for a integer that will be printed in hex