David Giacobbi
CPSC 260: Assignment #8 Responses

Problem 1:

movl	%eax	(%rsp)
movw	(%rax)	%dx
movb	$0xFF	%bl
movb	(%rsp, %rdx, 4)	%dl
movq	(%rdx)	%rax
movw	%dx	(%rax)

*answers are highlighted in yellow

Problem 2:

movb	$0xF	(%ebx)	%ebx is not an address register, can't be dereferenced
movl	%rax	(%rsp)	Instruction suffix and register ID do not match in size
movw	(%rax)	4(%rsp)	Memory references can't be both the source and destination
movb	%al	%sl	%sl does not exist in register library
movq	%rax	$0x123	Immediate/literal values cannot be destination
movl	%eax	$rdx	Wrong size for the destination operand
movb	%si	8(%rbp)	Instruction suffix and register ID do not match

*answers are highlighted in yellow

Problem 3 (Screenshots):

Shell × switch_case.s × +

switch_case.s

```
57  to_upper_case:
58    subb    $32, %bl      # make current character uppercase
59    movb    %bl, (%r9)    # store result as pointer
60    retq
61
62  # Lower case function
63  to_lower_case:
64    addb    $32, %bl      # make current character lowercase
65    movb    %bl, (%r9)    # store result as pointer
66    retq
67
68          .data
69  mystr:  .ascii "FOOBAR\n"
70          .equ mylen, (. - mystr)
71
```

Shell × +

```
~/cpsc260-hw8$ gcc -c -g switch_case.s
~/cpsc260-hw8$ ld switch_case.o -o switch_case
~/cpsc260-hw8$ ./switch_case
foobar
~/cpsc260-hw8$
```

Shell × switch_case.s × +

switch_case.s

```
57  to_upper_case:
58    subb    $32, %bl      # make current character uppercase
59    movb    %bl, (%r9)    # store result as pointer
60    retq
61
62  # Lower case function
63  to_lower_case:
64    addb    $32, %bl      # make current character lowercase
65    movb    %bl, (%r9)    # store result as pointer
66    retq
67
68          .data
69  mystr:  .ascii "foo_bar!\n"
70          .equ mylen, (. - mystr)
71
```

Shell × +

```
~/cpsc260-hw8$ gcc -c -g switch_case.s
~/cpsc260-hw8$ ld switch_case.o -o switch_case
~/cpsc260-hw8$ ./switch_case
FOO_BAR!
~/cpsc260-hw8$
```

Problem 4 (Screenshots):

sort.c × selection_sort.s × +

sort.c › main

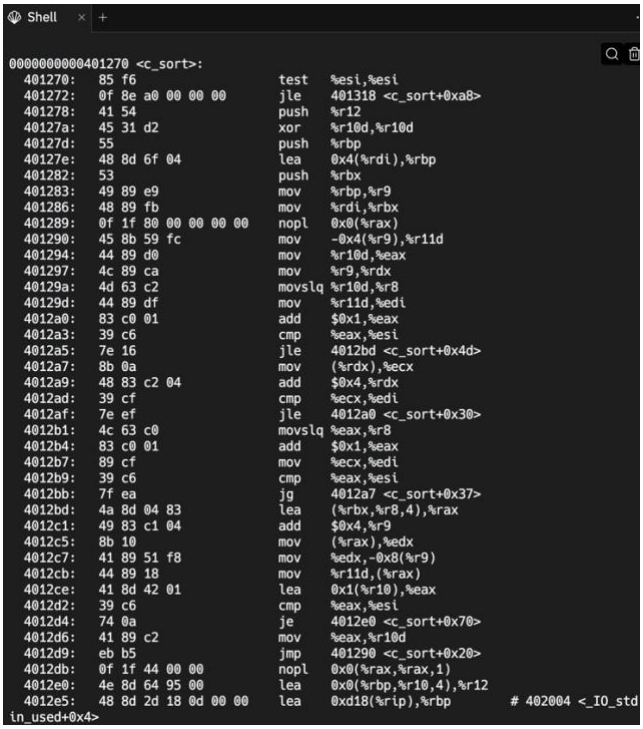
```
2
3   void selection_sort(int array[], int length);
4   void c_sort(int array[], int length);
5
6   int main(){
7
8     // Test arrays for both programs
9     int array1[] = {2, 5, 7, 2, 4, 5, 9, 1};
10    int array2[] = {1, 3, 9, 2, 4};
11    int array3[] = {8, 4, 1, 3, 9, 3, 5, 6, 0, 2};
12
13    // ARRAY 1 Assembly Sort
14    selection_sort(array1, 8);
15    for (int i = 0; i < 8; i++){
16      printf("%d ", array1[i]);
17    }
18    printf("\n");
19
20    // ARRAY 2 Assembly Sort
21    selection_sort(array2, 5);
22    for (int i = 0; i < 5; i++){
23      printf("%d ", array2[i]);
24    }
25    printf("\n");
26
27    // ARRAY 3 Assembly Sort
28    selection_sort(array3, 10);
29    for (int i = 0; i < 10; i++){
30      printf("%d ", array3[i]);
31    }
32    printf("\n");
33
34    return 0;
35  }
```

Shell × +

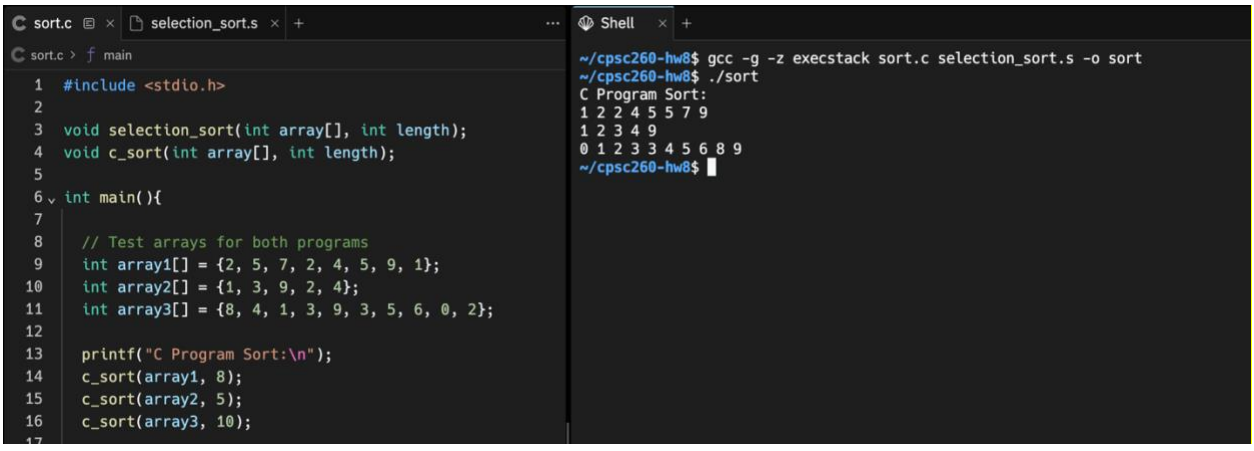
```
~/cpsc260-hw8$ gcc -g -z execstack sort.c selection_sort.s -o sort
~/cpsc260-hw8$ ./sort
1 2 2 4 5 5 7 9
1 2 3 4 9
0 1 2 3 3 4 5 6 8 9
~/cpsc260-hw8$
```

Problem 5 (Screenshots and Observations):

Object Dump Screenshots:



Testing Results Screenshot:



Observations between Object Dump and My Code:

One of the first observations that I made between my own code and the C-compiled assembly is that it decided to push registers onto the stack as they were needed as variables in the function. I think the reasoning behind this is that I had time to think about the structure of my function and what registers I would use ahead of time. Since the compiler is simply reading in lines of my C code, it does not have this luxury and, therefore, pushes available registers onto the stack before altering them.

Another observation was the compiler's use of the lea function when loading effective addresses. An interesting tactic that I noticed was the use of a negative shift in line 401290. It calls the dereferenced %r9 register with a -4 shift: -0x4(%r9), %r11d. I have never worked with a shift like this and I am curious to know how this could possibly optimize my own function.

Lastly, I noticed the use of a nopl assembly call, which I have not seen before. This would be another call to investigate its purpose as it was often associated with the dereferencing of a register on multiple occasions. Overall, though there are some instances of optimization, the object dump revealed that my own code and its implementation were roughly the same length and, given the restraints of selection sort, roughly follow the same path but maybe use different registers in completing the same task.