CSE 351 Section 4 – C and x86-64 Assembly

Hi there! Welcome back to section, we're happy that you're here ☺

What is Assembly?

Assembly language is a human-readable representation of machine code instructions (generally a one-to-one correspondence). Assembly is machine-specific because the computer architecture and hardware are designed to execute a particular machine code instruction set.

x86-64

x86-64 is the primary 64-bit instruction set architecture (ISA) used by modern personal computers. It was developed by Intel and AMD and its 32-bit predecessor is called IA32. x86-64 is designed for complex instruction set computing (CISC), generally meaning it contains a larger set of more versatile and more complex instructions.

For this course, we will utilize only a small subset of x86-64's instruction set and omit floating point instructions.

x86-64 Instructions

The subset of x86-64 instructions that we will use in this course take either one or two operands, usually `operation operand1, operand2`. Operands can be:

- Immediate: constant integer data (*e.g.* `$0x400`, `$-533`) or an address/label (*e.g.* `Loop`, `main`)
- Register: use the data stored in one of the 16 general purpose registers or subsets (*e.g.* `%rax`, `%edi`)
- Memory: use the data at the memory address specified by the addressing mode `D(Rb,Ri,S)`

The operation determines the effect of the operands on the processor state and has a suffix ("`b`" for byte, "`w`" for word, "`l`" for long, "`q`" for quad word) that determines the bit width of the operation. Sometimes the operation size can be inferred from the operands, so the suffix is omitted for brevity.

Control Flow and Condition Codes

Internally, condition codes (Carry, Zero, Sign, Overflow) are set based on the result of the previous operation. The `j*` and `set*` families of instructions use the values of these "flags" to determine their effects.

An *indirect jump* is specified by adding an asterisk (*) in front of a memory operand and causes your program counter to load the address stored at the computed address.

Procedure Basics

The instructions `push`, `pop`, `call`, and `ret` move the stack pointer (`%rsp`) automatically.

`%rax` is used for the return value and the first six arguments go in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` ("**D**iane's **S**ilk **D**ress **C**ost **$89**").

Exercises:

1. [CSE351 Au14 Midterm] What does the following code return?

```
movl   (%rdi), %eax           # %rdi -> x;  r = *x
leal   (%eax,%eax,2), %eax    # %rax -> r;  r = (*x) * 3
addl   %eax, %eax             #             r = (*x)*3 + (*x)*3
andl   %esi, %eax             # %rsi -> y;  r = ((*x)*6) & y
subl   %esi, %eax             #             r = (((*x)*6) & y) - y
ret
```

(((*x) * 6) & y) - y

2. [CSE351 Au15 Midterm] Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just the correctness.

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
        return *x;
}
```

```
happy:
    cmpq  %rdx, %rsi
    jle   .else
    leaq  (%rdx, %rsi), %rax
    ret
.else:
    movq  (%rdi), %rax
    ret
```

Multiple other possibilities (*e.g.* switch ordering of if/else clauses, replace `lea` with `mov`/`add` instruction pair).

3. Write an equivalent C function for the following x86 code:

```
mystery:
1    testl    %edx, %edx            # %edx is 3rd argument (z)
2    js       .L3                   # jump to .L3 if z<0
3    cmpl     %esi, %edx            # %esi is 2nd argument (y)
4    jge      .L3                   # jump to .L3 if y>=z
5    movslq   %edx, %rdx            # sign-extend 3rd argument (z)
6    movl     (%rdi,%rdx,4), %eax   # %rdi is 1st argument (x), calc *(x + z*4)
7    ret
.L3:
8    movl     $0, %eax              # return 0
9    ret
```

```
int mystery(int *x, int y, int z) {
    if (z >= 0 && z < y)
        return x[z];
    else
        return 0;
}
```

Notes:

- If either conditional is True, then we jump to the "else" clause, so in C we execute the "if" clause only when the complement of both of them are True.
- Line 6 indicates that the return type is 4 bytes (`int`). Line 8 is ambiguous since it zeros out the entire 8 bytes of `%rax`.
- Argument variable names are arbitrary. Based on usage, could perhaps have used `x`→`ar`, `y`→`n`, `z`→`k`.
- First argument had to point to `int` based on scale factor in Line 6. Both `int *x` and `int x[]` work.

4. [Adapted from CSE351 Wi16 Midterm] Suppose before the assembly below is executed, the value of `%rsp` is `0xFFFF8888`.

```
0x00002f:   pushq $7
0x000031:   pushq $5
0x000033:   addq  $2, 8(%rsp)
0x000039:   callq someOtherFunction
0x00003e:   ...
```

Immediately *after* the `callq` instruction executes:

a. What is the value of `%rsp` in hexadecimal?

The `push` and `call` instructions add to the stack (decrement `%rsp`). There are two `pushq` and one `callq` so `%rsp` has been decremented by 24 bytes. 24 = 0x18. 0xFFFF8888 – 0x18 = **`0xFFFF8870`**.

b. Fill in the contents of the stack from `%rsp` (your answer to part a) up to (but not including) `0xFFFF8888`. Fill in the boxes below using hexadecimal. You may not need all rows.

Address	Data
0xFFFF8888	<unknown>
0xFFFF8880	0x9
0xFFFF8878	0x5
0xFFFF8870	0x3e

Notes:

- Originally 7 is pushed onto the stack first, but then later gets 2 added to it.
- The return address is the address of the instruction *after* `callq`, which is 0x00003e in this case.