Assignment 1
x86-64 Assembly

Student Name: Your name, your.email@ntut.edu.tw
Student ID: Your student ID number,
Instructor: Jyun-Ao Lin, jalin@ntut.edu.tw

The goal of this assignment is to get some familiarity with x86-64 assembly language, by manually compiling small C programs.

An assembly code is written in a file with suffix `.s` and looks like this:

```
      .text
      .globl main
main:
      ...
      mov  $0, %rax       # exit code
      ret
      .data
      ...
```

You can compile and run such a program as follows:

```
gcc -g file.s -o file
./file
```

(Add the option `-no-pie` if you use `gcc` version 5 or later.)

When needed, you can use `gdb` to execute your program step by step. Use the following commands

```
gdb ./file
(gdb) break main
(gdb) run
```

and then execute one step with command `step`. More information in this tutorial.

This page by Andrew Tolmach provides some information to write/debug x86-64 assembly code. These notes on x86-64 programming are really useful.

Exercise 1: Printing using `printf` (10 pts)

Compile the following C program:

```
#include <stdio.h>

int main() {
    printf("n = %d\n", 42);
    return 0;
}
```

To call the library function `printf`, we pass its first argument (the format string) in register `%rdi` and its second argument (here the integer 42) in register `%rsi`, as specified by the calling conventions. We must also set register `%rax` to zero before calling `printf`, since it is a variadic function (in that case, `%rax` indicates the number of arguments passed in vector registers —

here none).
The format string must be declared in the data segment (`.data`) using the directive `.string` that adds a trailing 0-character.

Exercise 2: Arithmetic Expressions (10 pts)

Write assembly programs to evaluate and display the results of the following arithmetic expressions:

- `4 + 6`
- `21 * 2`
- `4 + 7 / 2`
- `3 - 6 * (10 / 5)`

The expected results are

```
10
42
7
-9
```

To display an integer, you can use the solution of Exercise 1.

Exercise 3: Boolean Expressions (10 pts)

Taking the convention that the integer 0 represents the Boolean value *false* and any other integer represents the value *true*, write assembly programs to evaluate and display the results of the following expressions (you must display true or false in the case of a Boolean result):

- `true && false`
- `if 3 <> 4 then 10 * 2 else 14`
- `2 = 3 || 4 <= 2 * 3`

The expected results are

```
false
20
true
```

It will be useful to write a `print_bool` function to display a boolean.

Exercise 4: Global Variables (10 pts)

Write an assembly program that evaluates the following three instructions:

```
    let x = 2
    let y = x * x
    print (y + x)
```

The variables x and y will be allocated in the data segment. The expected outcome is 6.

Exercise 5: Local Variables (10 pts)

Write an assembly program that evaluates the following program:

```
    print (let x = 3 in x * x)
    print (let x = 3 in (let y = x + x in x * y) + (let z = x + 3 in z / z))
```

We will allocate the variables x, y and z in the stack. The expected result is

```
    9
    19
```

Exercise 6: Integer Square Root (20 pts)

Compile the following C program:

```
#include <stdio.h>

int isqrt(int n) {
    int c = 0, s = 1;
    while (s <= n) {
        c++;
        s += 2*c + 1;
    }
    return c;
}

int main() {
    int n;
    for (n = 0; n <= 20; n++)
    printf("sqrt(%2d) = %2d\n", n, isqrt(n));
    return 0;
}
```

Try to do the following:

- a single branching instruction per loop iteration;
- a single instruction to compile the assignment `s += 2*c + 1`.

Exercise 7: A Slightly More Complex (and More Interesting) Program (30 pts)

We are now considering the compilation of a C program to solve the following problem: given the following 15×15 integer matrix

```
  7  53 183 439 863 497 383 563  79 973 287  63 343 169 583
627 343 773 959 943 767 473 103 699 303 957 703 583 639 913
447 283 463  29  23 487 463 993 119 883 327 493 423 159 743
217 623   3 399 853 407 103 983  89 463 290 516 212 462 350
960 376 682 962 300 780 486 502 912 800 250 346 172 812 350
```

```
870 456 192 162 593 473 915  45 989 873 823 965 425 329 803
973 965 905 919 133 673 665 235 509 613 673 815 165 992 326
322 148 972 962 286 255 941 541 265 323 925 281 601  95 973
445 721  11 525 473  65 511 164 138 672  18 428 154 448 848
414 456 310 312 798 104 566 520 302 248 694 976 430 392 198
184 829 373 181 631 101 969 613 840 740 778 458 284 760 390
821 461 843 513  17 901 711 993 293 157 274  94 192 156 574
 34 124   4 878 450 476 712 914 838 669 875 299 823 329 699
815 559 813 459 522 788 168 586 966 232 308 833 251 631 107
813 883 451 509 615  77 281 613 459 205 380 274 302  35 805
```

determine the maximal sum we can obtain using exactly one element per row and per column. (This is the Problem 345 from project Euler)

The C program `matrix.c` (in Teams) contains a solution, to be read carefully. This solution uses two main ingredients:

1. Generalize the problem to a subset of rows and columns. This subset is defined using two integers `i` and `c`:

   - We only consider rows `i..14` ;
   - We only consider columns `j` for which the bit `j` in the integer `c` is 1. (It is an invariant that `c` has exactly 15-`i` bits that are set.)

   The call `f(i, c)` returns the maximal sum for the subset defined by `i` and `c`.

2. We use memorization, to avoid recomputing `f(i, c)` several times. For this purpose, an array `memo` is declared. We store the result of `f(i, c)` at index `c << 4` i—, when it is computed, and 0 otherwise.

Representing the Matrix

In the C program, the matrix `m` is declared as follows:

```
const int m[N][N] = { { 7, 53, ..., 583 }, { 627, 343, ... }, ... };
```

In the memory layout, integers are stored consecutively, by rows. Each integer is stored on 32 bits and thus the matrix `m` uses 900 bytes in total. The integer `m[i][j]` is located at address `m + 4 * (15*i + j)`.

We provide a file `matrix.s` (c.f. Teams) that contains static data for the matrix `m`, as well as space for the array `memo`. The latter is initialized with zeros, and thus can be allocated in section `.bss` so that it does not increase the size of the executable unnecessarily.

Compiling the program

Compile functions `f` and `main`. Regarding function `f`, we need to allocate registers for local variables `key, r, s`, etc. If we choose callee-saved registers, we need to restore them before returning. If we choose caller-saved registers, we weed to restore them after a call (if we need their value after the call).

Be careful: to compute `1 << j`, one must do a shift whose size is not a constant. To do so, the operand of instruction `sal`, which is `j` here, must be placed in the byte register `%cl`. For this reason, it is a good idea to allocate variable `j` in register `%rcx`.