Assembly and Machine Language - Fall 1397 (2018) Final Exam	**Instructor: B. Nasihatkon**	دانشگاه صنعتی خواجه نصیرالدین طوسی K. N. TOOSI UNIVERSITY OF TECHNOLOGY
Name:	**ID:**	**Dey 1397 - January 2019**

- All Assembly/C programs are written for a 32-bit x86 architecture.

Question 1 (20 points) What does the following C code print? Explain why. Do not forget the new lines. Assume 4-byte `int` type.

```
int array[] = {10,20,30,40,50,60,70,80};
int size = sizeof(array)/sizeof(int);

asm volatile("lea esi, [edi+4] ;"
             "mov ebx, [edi]   ;"
              "cld              ;"
              "rep movsd        ;"
              "mov  [edi], ebx  ;"
              :
              :"D"  (array), "c"  (size - 1)
              :"memory","ebx","eax");


  for (int i = 0; i < size; i++)
    printf("%d\n", array[i]);
```

Question 2 (25 points)

Polar coordinates (ρ, θ) of a 2D point can be converted to Cartesian coordinates (x, y) according to

$x = \rho \cos\theta$

$y = \rho \sin\theta$

In the following assembly program, the radial coordinate ρ and angular coordinate θ are stored in the data segment as *double precision* floating points with labels **`rho`** and **`theta`** respectively. Write an assembly program to compute the *x* and *y* Cartesian coordinates of the point, store them in the memory locations labeled `x` and `y`, and then prints the point with a single `printf` function call equivalent to the following function call in C:

`printf("(%f,%f)\n", x, y)`

Notice that, here, θ is stored in *degrees*. To use the assembly instructions **`fsin, fcos,`** and **`fsincos`** you need to first convert it to radians (radians = degrees * π / 180). You may use the following assembly instructions.

`fldpi` pushes π on the FPU register stack.

`fsincos` Computes sin(ST0) and cos(ST0), replaces ST0 with the sine and pushes the cosine on the FPU stack.

You can use `fsin` and `fcos` instructions as well, but you lose points. Define new data in the data segment if needed.

label	command	arguments
segment .data		
rho: **theta:** **x:** **y:** **format:**	dq 10.0 dq 150.0 dq 0.0 dq 0.0 db "(%f,%f)", 10, 0	
segment .text extern **printf**		

label	command	arguments

Question 3 (28 points)

The following C code generates a random number between 0 and 1023 as *control code* and then asks the user to enter a number as the response. The true response is some sort of hash function (named **hash**) applied to the control code. You can see a sample run of the compiled program on the right. We only have access to the compiled program and the following C code, but not the source code of the **hash** function. Fortunately, the executable is not stripped and we could easily disassemble it using GDB (see below).

```
int main() {
  unsigned int control_code, response;

  srand(time(NULL));
  control_code = rand() % 1024;

  printf("Control Code: %u\n", control_code);

  printf("Response: ");

  scanf("%u", &response);

  if (response == hash(control_code))
    puts("Correct!");
  else
    puts("Incorrect!");

  return 0;
}
```

```
Control Code: 800
Response: 1234
Incorrect!
```

```
(gdb) set disassembly-flavor intel
(gdb) disassemble hash
Dump of assembler code for function hash:
   0x08048650 <+0>: mov    eax,DWORD PTR [esp+0x4]
   0x08048654 <+4>: lea    eax,[eax+eax*2+0x64]
   0x08048658 <+8>: and    eax,0x3ff
   0x0804865d <+13>:     lea    eax,[eax+eax*4]
   0x08048660 <+16>:     lea    eax,[eax+eax*1+0xc8]
   0x08048667 <+23>:     and    eax,0x3ff
   0x0804866c <+28>:     ret
```

A) Explain what the function **hash** does. Then, write an equivalent C code defining the this function. You are only allowed to use the arithmetic operators `+`, `-`, `*`, `/`, and `%`. You miss points by using the bitwise AND operator &. (23 points)

`unsigned int hash(`	**Explanation**

B) If the random control code is **800**, what response code the user must enter? Why? (5 points)

Question 4 (27 points)

The following C function **`myPuts`** receives a string as input and prints it. It also returns the length of the string as the return value. Write the equivalent assembly code to define the function **`myPuts`**. The definition must be defined recursively. It also has to be callable from within C, so observe all C calling conventions. To print a single character you must call the function **`putchar`** from the C standard library. Do not forget to declare the **global** and **extern** symbols. Rembeber, `putchar` takes an integer as its argument (not a character): `int putchar(int c);` Assume 4-byte **`int`** type.

```
int myPuts(char *s) {
  if (*s == 0) {
    putchar('\n');
    return 0;
  }

  putchar(*s);
  return 1 + myPuts(s+1);

}
```

label	command	arguments
extern **global**		
myPuts:		

label	command	arguments