COMP2611 Spring 2017 Homework #2

(Due Monday Apr 3, 5:00PM)

IMPORTANT Notes:

- This is an individual assignment; all works must be your own.
- Add your code under TODOs in the skeleton code. Don't change any other code provided. Keep the skeleton file name unchanged.
- **Follow the usage of registers as specified in the skeleton**, otherwise the provided functions may not be able to work properly.
- Zip all MIPS assembly files in a single *<your_stu_id>.zip* file.
- Submit your work via CASS (http://cssystem.cse.ust.hk/UGuides/cass/student.html). Multiple submissions are allowed but only the last submission before the deadline will be graded.
- **No LATE submissions will be accepted.** Plan well and don't rush to submit in the last minute.
- **Your submitted program must be able to run under MARS, otherwise it will not be marked.**

Question 1: Caesar's cipher (10 marks)

Caesar's cipher encrypts a plain text by replacing each of its character with another character according to a fixed shift amount value **(**or **key)**. Refer to https://en.wikipedia.org/wiki/Caesar_cipher for further details.

Write a full MIPS program that takes **a string** (plain text) and **a positive integer** (key) and then outputs the encrypted string accordingly. Encrypt only the letters and keep the spaces in the text unchanged.

For simplicity, assume all user inputs are always positive from 1-25. Refer to `Caesar.s` for more details. Read comments in skeleton and follow its instructions.

Sample run of the program in MARS:

```
Please enter the message all in capital letters: AMAZING STUDENTS
Please enter the shift amount: 3
The encrypted text is: DPDCLQJ VWXGHQWV
-- program is finished running --
```

Question 2: Unsigned integer multiplication by bit shifting (20 marks)

The Shift-and-add multiplication approach is similar to the multiplication performed by using paper and pencil. The following example illustrates the multiplication procedures using two unsigned 4-bit numbers, $1000_{(2)}$ and $1101_{(2)}$. Without loss of generality we denote the first number the **multiplicand**, and the second number **multiplier**.

```
Multiplicand       1000 ×    (stored in $s0 of your program)
Multiplier         1101      (stored in $s1 of your program)
              --------------
                   1000    (multiplicand × 1, shift the product 0 bit left,  $s2=1000 )
                  0000     (multiplicand × 0, shift the product 1 bit left , $s2=$s2+00000)
                 1000      (multiplicand × 1, shift the product 2 bits left, $s2=$s2+100000)
                1000       (multiplicand × 1, shift the product 3 bits left, $s2=$s2+1000000)
              --------------
                1101000    (done! $s2=1101000)
```

Complete the "`multi.s`" skeleton provided according to the shift-and-add multiplication procedures illustrated above. Store the multiplicand in `$s0` and the multiplier in `$s1`. Accumulate the result according to the above example step-by-step into `$s2`, output `$s2` in hexadecimal form (using syscall 34, check the skeleton for the details) after every step as shown below in the sample run. You program should be able to handle multiplications of two unsigned 8-bit integers and you do not need to worry about overflows. You may assume the user inputs are always valid.

Hint: To extract the bits of the multiplier one by one, you put 1 into `$s3` and extract the first bit of multiplier into `$t0` using "`and $t0,$s1,$s3`" in the first iteration. Then in the next iteration you shift $s3 by one bit to the left ("`sll $t3, $t3, 1`") and then use "`and $t0,$s1,$s3`" to extract the second bit of the multiplier into `$t0`. By referring to `$t0` in each iteration, you will know what to accumulate to `$s2` in the program.

Sample run of the program in MARS:

```
Please enter the Multiplicand [0-255]: 255
Please enter the Multiplier   [0-255]: 211
The result at the current iteration is: 0x000000ff
The result at the current iteration is: 0x000002fd
The result at the current iteration is: 0x000002fd
The result at the current iteration is: 0x000002fd
The result at the current iteration is: 0x000012ed
The result at the current iteration is: 0x000012ed
The result at the current iteration is: 0x000052ad
The result at the current iteration is: 0x0000d22d
The product is: 0x0000d22d
-- program is finished running --
```

Question 3: Sort random array (20 marks)

The following C++ program generates a random array and then sorts it in ascending order. Translate it into MIPS program. The translation doesn't need to be 100% identical but should follow the same sorting algorithm as described.

For simplicity, the MIPS program assumes array `A[]` has a fixed size of 10. You can use the provided procedure `random_number_generate to` generate 10 random integers and store them in `A[]`. Refer to `randArraySort.s` for more details. Read comments in skeleton and follow its instructions. Note that we do the sorting by calling the sort() function in the C/C++ code. In the skeleton program, we do it in the main body of the program.

```
#include <iostream>       /* cout */
#include <cstdlib>      /* srand, rand */
#include <ctime>        /* time */

using namespace std;


void randArray(int a[], int n){

        srand (time(NULL)); //random seed to init the random number generator

        for (int i=0;i<n-1;i++) {
               a[i]=rand() % 100 + 1; //random numbers in the interval [1,100]
        }
}

void Sort(int a[], int n){       /*n is the size of the array a */
       int i,j,temp;
       for (i=0; i<n-1; i++) {
         for (j=0; j<n-1-i; j++)
           if (a[j+1] < a[j]) {   /* compare the two neighbors */
             temp = a[j];         /* swap a[j] and a[j+1]      */
             a[j] = a[j+1];
             a[j+1] = temp;
         }
       }
}

int main(){
       int A[10];

       randArray(A,10);
       Sort(A,10);

       for (int i=0;i<9;i++){
              std::cout<<A[i]<<" ";
       }
       return 0;
}
```

Sample run of the program in MARS:

```
The original list of random numbers are
41 24 46 72 19 70 27 60 51 4

The ascending sorted numbers are
4 19 24 27 41 46 51 60 70 72

-- program is finished running --
```