# COMP2611 Spring 2022 Homework #3

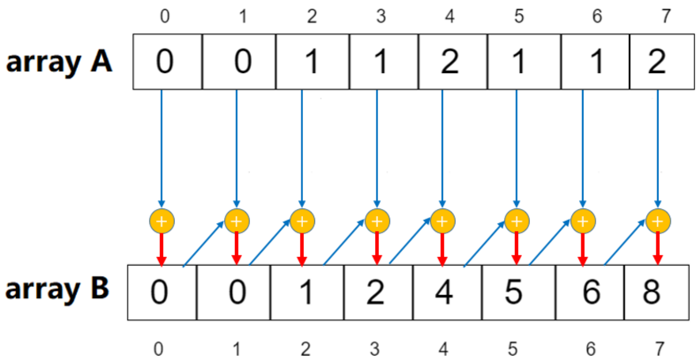
**(Deadline 11:55pm, Tuesday April 19, 2022 HKT, UTC+8)**

**Notes:**

- This is an individual assignment; all works must be your own. You can discuss with your friends but never show your code to others.
- Write your code in given MIPS assembly skeleton files. Add your own code under TODOs in the skeleton code. Keep other parts of the skeleton code unchanged.
- Make procedure calls with the registers as specified in the skeleton, otherwise the provided procedures may not work properly. Preserve registers according to the MIPS register convention on slide 76 of the ISA note set.
- Zip the three finished MIPS assembly files into a single zip file, *<your_stu_id>.zip* file (without the brackets). Do not change names of the given skeleton files.
- To submit, first find the Canvas page of COMP2611, homework 3, and then upload the "*<your_stu_id>.zip*" file. You can upload for as many times as you like, only the latest one before the deadline will be marked.
- **Solutions of this homework will be posted at the course web right after the deadline; so no late submission will be accepted.**
- **Your submitted program must be able to run under MARS, otherwise it will not be marked**.

## Question 1: Cumulative Sum Array (20 marks)

An array `B[]` is called the cumulative sum array of array `A[]`, if `B[]` has the same length as `A[]`, and `B[i]= A[0] + A[1] + ... A[i-1] + A[i]`. The picture below illustrates the idea:






Refer to the following C++ program for the details in calculating and outputting a cumulative sum array for a user-specified **integer array** `A[]`.



Your task is to implement the `AccumulateArray` MIPS procedure in skeleton file `AccumulateArray.asm`. The procedure should implement the same functionality as the `AccumulateArray()` function in the C++ program.



**Add your own code under the "TODO" in the skeleton code. Follow the instructions specified in the "TODO". DO NOT modify other parts of the skeleton code. NO new procedure is allowed**. For simplicity you can assume all the input elements of array `A[]` are valid signed integers. You can also assume that there is no overflow in all the cumulative sums.

**C++ program**

```cpp
#include <iostream>
using namespace std;

const int SIZE = 10; // assume 10 elements

// A[] is the source array
// B[] is the cumulative sum array of A[]
void AccumulateArray(int A[], int B[], int size)
{
    // copy A[0] to B[0]. prepare for the loop.
    B[0] = A[0];

    // use last B[i] and A[i+1] to caculate B[i+1]
    for (int i = 0; i < size - 1; i++)
    {
        B[i + 1] = B[i] + A[i + 1];
    }
}

int main()
{
    int A[SIZE];
    int B[SIZE] = {0}; // init all elements to 0
    int count = 0;
    int i = 0;

    cout << "Please enter integers in array A[] one by one:" << endl;
    for (i = 0; i < SIZE; i++)
    {
        cout << "A[" << i << "]: ";
        cin >> A[i];
    }

    AccumulateArray(A, B, SIZE);

    cout << "Here is the accumulative array of A:" << endl;
    for (i = 0; i < SIZE; i++)
    {
        cout << B[i] << ' ';
    }
    return 0;
}
```

**A sample execution of the program in MARS**

```
Please enter integers in array A[] one by one:
A[0]: 1
A[1]: 2
A[2]: 3
A[3]: 4
A[4]: 5
A[5]: 6
A[6]: 7
A[7]: 8
A[8]: 9
A[9]: 10
Here is the accumulative array of A:
1 3 6 10 15 21 28 36 45 55
-- program is finished running --
```

## Question 2: Skew-Symmetric Matrix (20 marks)

A square matrix `A` is called a skew-symmetric matrix if it satisfies the following condition:

$$A^T = -A$$

That is, a square matrix `A` is skew-symmetric if its transpose is equal to its negative. The following illustrates an example of a skew-symmetric square matrix:

$$A = \begin{bmatrix} 0 & 2 & -45 \\ -2 & 0 & -4 \\ 45 & 4 & 0 \end{bmatrix} \qquad A^\top = \begin{bmatrix} 0 & -2 & 45 \\ 2 & 0 & 4 \\ -45 & -4 & 0 \end{bmatrix} \qquad A + A^\top = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

If we denote by `a`$_{\texttt{ij}}$ the element in the `i`$^{\texttt{th}}$ row and `j`$^{\texttt{th}}$ column of a matrix `A`, then the following is a useful property of the skew-symmetric matrix:

$$a_{ij} = -a_{ji}\,, \qquad \forall a_{ij} \in A$$



We will be using the above condition to check if a user input matrix is skew-symmetric.



Your task is to implement `IsSkewSymmetric` MIPS procedure in skeleton file `IsSkewSymmetric.asm`. The procedure should implement same functionality as the `IsSkewSymmetric()` function in the C++ program



**Add your own code under the "TODO" in the skeleton code. Follow the instructions specified in the "TODO". DO NOT modify other parts of the skeleton code. NO new procedure is allowed**. For simplicity you can assume the biggest user inputted matrix size is `5x5` (i.e. 5 rows, 5 columns just like in the `C++` program). You also can assume the matrix `A` is stored in the 1-D array `A[]` in row order in MIPS (under the label "`A`").

For example, the following matrix:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}$$

is stored in a 1-D array in the row order in MIPS as:

$$A[\,] = \{a_{0,0}, a_{0,1}, \dots, a_{0,n-1}, a_{1,0}, a_{1,1}, \dots, a_{1,n-1}, \dots, a_{n-1,0}, a_{n-1,1}\;, \dots, a_{n-1,n-1}\}$$

**C++ program**

```
#include <iostream>
using namespace std;

// do multiplication using addition
// this is provided to you in the MIPS skeleton for your calculations
int multiply(int a, int b)
{
    int result = 0;
    for (int i = 0; i < b; i++)
    {
        result = result + a;
    }
    return result;
}

// A[] is holding the square matrix to be checked,
// n is the size of the matrix
int IsSkewSymmetric(int A[], int n)
{
    int a_ij = 0; // a_ij of matrix A
    int a_ji = 0; // a_ji of matrix A
    int idx = 0;  // index for accessing an element of matrix A

    // check to see if a_ij == -a_ji, for all a_ij in the matrix,
    // where i is row number, j is column number
    // assume a_ij is located at a[i*row_size+j]
    for (int i = 0; i < n; i++)
    {
        // just need to check through roughly half of the elements
        for (int j = i ; j < n; j++)
        {
            //calculation the position of the element a_ij
            idx = multiply(i, n);
            idx = idx + j;
            a_ij = A[idx]; // a_ij
            //calculation the position of the element a_ji
            idx = multiply(j, n);
            idx = idx + i;
            a_ji = A[idx]; // value of a_ji
            // check if SkewSymmetric condition satisfied
            if (a_ij + a_ji != 0)
            { // not satisfied
                return 0;
```

```
            }
        }
    }
    return 1;
}

int main()
{
    int A[25]; // Array for holding the square matrix
    int n;     // size of the square matrix
    int idx;   // index to for storing a_ij of the matrix

    cout << "Please enter the size of your matrix:" << endl;
    cin >> n;
    cout << "Please enter integers in array A[] one by one:" << endl;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << "A[" << i << "]";
            cout << "[" << j << "]: ";
            idx = multiply(n, i);
            idx = idx + j; // calculate the index of element a_ij in array A[]
            cin >> A[idx];
        }
    }
    cout << "Is it a skew-symmetric matrix ? ";
    if (IsSkewSymmetric(A, n) == 0)
    {
        cout << "NO." << endl;
    }
    else
    {
        cout << "YES." << endl;
    }
}
```

**Two sample executions of the program in MARS**

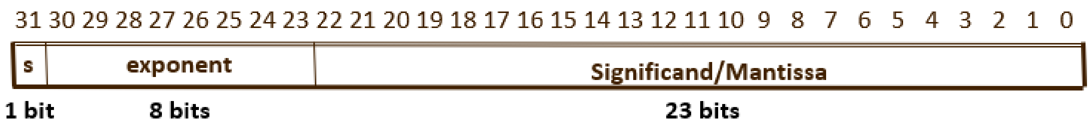
```
Please enter the size of your matrix:
3
Please enter integers in array A[] one by one:
A[0][0]: 0
A[0][1]: 2
A[0][2]: -45
A[1][0]: -2
A[1][1]: 0
A[1][2]: 4
A[2][0]: 45
A[2][1]: -4
A[2][2]: 0
Is it a skew-symmetric matrix ? YES.
-- program is finished running --
```



```
Please enter the size of your matrix:
2
Please enter integers in array A[] one by one:
A[0][0]: 1
A[0][1]: 2
A[1][0]: -2
A[1][1]: 1
Is it a skew-symmetric matrix ? NO.
-- program is finished running --
```

## Question 3: IEEE-754 Floating Point Number Decoder (20 marks)

Recall the following IEEE-754 single precision representation scheme:




The value stored in the representation scheme could be calculated using:

$$x = (-1)^S \times (1 + mantissa) \times 2^{Exponent-Bias}$$

Write a program to get user input for an IEEE-754 single precision number. The program will **extract the integer part** of the number and put it into the **signed integer** datatype (i.e. a 32-bit register in MIPS) before outputting it. For simplicity, we assume all the user inputted numbers belong to **normalized cases** (i.e. exponents are in the range `[1, 254]`).



Your task is to implement the `FloatDecoder` MIPS procedure in skeleton file `FloatDecoder.asm`. The procedure should implement the same functionality as the `FloatDecoder()` function in the C++ program.





**Add your own code under the "TODO" in the skeleton code. Follow the instructions specified in the "TODO". DO NOT modify other parts of the skeleton code. NO new procedure is allowed**. For simplicity you may assume that the user will always enter a number with exponent in the range `[1, 254]`, and the integer part of the number will not overflow the 32-bit signed integer datatype.

**C++ program**

```cpp
#include <iostream>
#include <cstring>
using namespace std;


// convert the 32-bit input binary sequence into a 32-bit unsigned int
// container “IEEE754EncodedNum”
unsigned int ToRegister(char code[])
{
    unsigned int IEEE754EncodedNum = 0;

    // getting the first 31 bits into the 32-bit int container
    for (int i = 0; i < 31; i++)
    {
        if (code[i] == '1')
        {
            IEEE754EncodedNum += 1;
        }
       IEEE754EncodedNum = IEEE754EncodedNum << 1;
    }

    // the rightmost bit, no need to shift
        if (code[31] == '1')
        {
            IEEE754EncodedNum += 1;
        }

    return IEEE754EncodedNum;
}

int FloatDecoder(unsigned int IEEE754EncodedNum)
{
    // exponent_mask is equivalent to the pattern 0111 1111 1000 0000 0000 0000 0000 0000
    // it is the "mask" for extracting the exponent (using bitwise AND)
    int exponent_mask=0x7f800000;
    // mantissa_mask is equivalent to the string 0000 0000 0111 1111 1111 1111 1111 1111
    // it is the "mask" for extracting the mantissa (using bitwise AND)
    int mantissa_mask=0x007FFFFF;
   // sign_mask is equivalent to the string 1000 0000 0000 0000 0000 0000 0000 0000
    // it is the "mask" for extracting the sign bit (using bitwise AND)
    int sign_mask=0x80000000;
```

```
int sign; // the value in the sign field
int exponent; // the value in the exponent field
int mantissa; // the mantissa in the mantissa field
int real_exponent; // the original exponent in the normalized expression
int shift_required; // shift required to get mantissa digits in the right places
int integer_part = 0; // the integer part of the number

mantissa = IEEE754EncodedNum & mantissa_mask;// get the mantissa
mantissa += 1 << 23;                          // add the implicit 1
// get the exponent stored in the IEEE scheme
exponent = (IEEE754EncodedNum & exponent_mask) >> 23;
// get the original exponent of the number
real_exponent = exponent - 127;

if (real_exponent >= 23)
{ // left shift amount based on the real_exponent value
  // note that the original mantissa is now occupying bits 0-22 in the mantissa
  // variable, so the mantissa variable is effectively holding the original mantissa
  // that has *already been shifted to the left by 23 bits*.
  // we need to see how many extra bits to *shift to the left*
  // the extra number of bits to shift is in the shift_required variable
    shift_required = real_exponent - 23;
    integar_part = mantissa << shift_required;
}
else
{   // real_exponent less than 23,
     // but the mantissa variable is holding the original
     // mantissa that has *already been shifted to the left by 23 bits*.
     // need to *shift to the right* to correct that
    shift_required = -1*(real_exponent - 23);

    // all the 24 digits (1 digit of implicit 1, and 23 digits of mantissa)
    // are shifted away, so the integer part is 0
    if (shift_required>24){
        integar_part=0;
    }
    else{// still has some digits remaining, shift to get the integer part
        integar_part = mantissa >> shift_required;
    }
}

sign = (IEEE754EncodedNum & sign_mask)>>31;
```

```
    if (sign == 0)
    {
        return integar_part;
    }
    else
    {
        integar_part = 0 - integar_part;
        return integar_part;
    }
}

int main()
{
    char code[32]; //char array to hold the input IEEE754 binary string

    // unsigned int type to get the 32 bits encoding in IEEE754
    // unsigned int is needed otherwise C++ right shift will do *sign extension*
    unsigned int IEEE754EncodedNum = 0;
    int integer_part;           //the extracted integer part
    cout << "Please enter the binary representation of your single prec
ision float number :" << endl;
    cin >> code;
    // convert the inputted binary string into a 32 bit container
    IEEE754EncodedNum = ToRegister(code);
    // decode the integer part of the IEEE754 encoded value
    integer_part = FloatDecoder(IEEE754EncodedNum);
    cout << "The integer part is :" << endl;
    cout << integer_part;
    return 0;
}
```

**Three sample execution of the program in MARS**

```
Please enter the binary representation of your single
precision float number :
11000100001011000010001000000000
The integer part is :
-688
-- program is finished running --
```

```
Please enter the binary representation of your single
precision float number :
01000100001011000010001000000000
The integer part is :
688
-- program is finished running --
```



```
Please enter the binary representation of your single
precision float number :
00000011100000000000000000000000
The integer part is :
0
-- program is finished running --
```