Homework 8.a: Subroutines (part a)

Single (input/output) parameter: In this part we use the Accumulator (AC) to pass the single parameter to the subroutine as an input parameter and from the subroutine as return parameter.

Note: in ALL implementations of a subroutine:
- You are **not** allowed to use the names of variables defined for main() in the subroutine and **vice versa**!
- Write the high-level algorithm first.
- Use the program skeleton where it is given and stick to the given names.
- Comment your programs properly!

Warm up: A subroutine with the signature: ***signed short int Absolute (signed short int a)***;
This subroutine takes one parameter as input – a **signed short integer** (16-bit) using 2's complement representation and returns its absolute value:

a. Write the high-level algorithm of the **main**() part that calls the subroutine and the subroutine in a pseudo high-level language.

b. Implement the algorithm in Mano CPU Assembly language and test them with different values. .

Note: in the following questions, all integers are 16 bit. For the sake of brevity, instead of writing:
- **signed short int** → **signed int**
- **unsigned short int** → **unsigned int**

1. You are given the following problem:

```
main ( ) { // the terminating element of both arrays is a negative number
        signed int Array1[] = {11,2,33, …, -999};
        signed int Array2[] = {11, 1, 2,13,1, 15, 2…, -999};
        unsigned int Ar1_count, Ar2_count;

        Ar1_count = InitArray(Array1);    // initialise Array1
        Ar2_count = InitArray(Array2);    // initialise Array2
}

// the subroutine initialises the array's elements to 0
unsigned InitArray( int *ArrPtr ) {
        ?
}
```

Can also be written as: **unsigned InitArray**(ArrPtr[])

Input Parameter: Address of the array

Implement the following in Mano CPU Assembly language AFTER writing the high-level algorithm:
unsigned int Arr(int *ArrPtr***)***: the subroutine initialises the array to 0 and ALSO counts and returns the number of the elements in the array. It then stores the result in an appropriate local variable, for example:
Ar1_count = **InitArray**(Array1);

The parameters:
- Array1 and Array2 are the addresses of the start of the array and are passed in the AC (**NOT as a Global variable**!).

Suggested skeleton

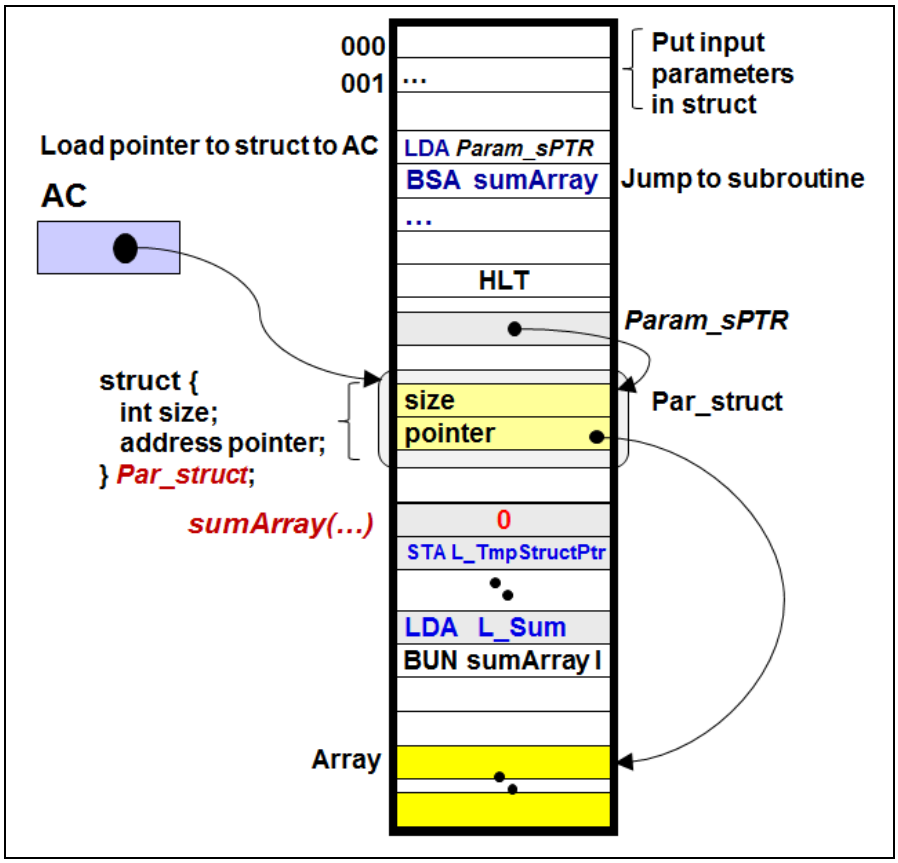
```
// main data
Ar1_Start,      HEX     100   // pointer to start of Array1
Ar1_count,      DEC     0     // Array1 size - calculated by subroutine Init_Array
Ar2_Start,      HEX     200   // pointer to start of Array2
Ar2_count,      DEC     0     // Array2 size - calculated by subroutine Init_Array
//
                ORG     100
Array1,         DEC     3           //
                DEC     6
                DEC     9
                DEC     -999        // Array Terminating value

                ORG     200
Array2,         DEC     11          //
                DEC     12
                DEC     13
                DEC     77
                DEC     7
                DEC     -999        // Array Terminating value

// Subroutine InitArray data
ArrPtr,         HEX 0   // temporary pointer to Array
Count,          DEC 0   // where to store the return parameter of the subroutine – the size of array
```

2. Question 2 uses the following mechanism for transferring more than 1 argument:

Multiple (input/output) parameters: in this part we pass a pointer to a STRUCT to the subroutine as input parameter, and if more than a single return parameter is needed we also pass it in a STRUCT.



You are given the following problem:

```
main ( ) {
      signed length Arr_Size = 6;  // The length of the arrays is given
       int result;
      signed int Array[] = {1, 2, 3, 4, 5, 6};
      Address Arr_Start = @Array;

      result = SumArray( ASize, Array);
  }
// the subroutine SumArray calculates & returns the sum of the elements of the array:
    short int SumArray (signed int Size,  int *ArrPtr ) {
        unsigned short int Sum = 0;
          ...
        FOR( ? )
        DO
            ?
        OD;
        return Sum;
    }
```

Input Parameters: size of the array & the addresses of the array – passed in struct

Output Parameter: sum of the elements of the array. Passed back through the AC

The **input parameters** are passed as a pointer to a struct of 2 fields:

- size is the length of the array.
- @Array is the addresses of the array.

The **return parameter** is passed through the AC.

Implement the algorithm in Mano's CPU Assembly language AFTER writing the high-level algorithm. Do not forget to comment the code with high-level comments.

You can use the template for solving exercise 2 in the following page.

The initialisation of the struct Par_struct has to be done **dynamically**!
You can use the following template for solving exercise 2:

```
// main
Main,    LDA     Param_sPTR      //
                                 // use the pointer to the Struct
                                 // prepare parameter Array_size
                                 // Par_struct.size = A_size;

                                 // prepare parameter address of Array
                                 // Par_struct.pointer = Ar_Start;
                                 // AC = @Par_struct;
         BSA     SumArray        //
         STA     Result          // result = SumArray( size, Array[ ] );
         HLT
// main data
Result,         DEC     0       //

Ar_Start,       HEX     100     // pointer to start of Array
A_size,         DEC     4       // size of the array
//
                ORG     100
Array,          DEC     1       //
                DEC     2
                DEC     3
                DEC     4

TempPtr,        HEX     0       // temporary pointer
// struct used to pass 2 parameters to subroutine SumArray
Param_sPTR, HEX     200     // pointer to Par_struct
                ORG     200     // parameter struct
// Struct containing:
Par_struct,     HEX     0       //   size of arrays
                HEX     0       //   pointer to array
// end of main data


                ORG     300
// Subroutine SumArray
SumArray,       HEX     0               // signed int SumArrays(signed int Size, signed int Array[]) {
                STA     L_TmpStructPtr  //


End_Loop,       LDA     L_Sum           //     return Sum;
                BUN     SumArray  I     // }

// local data of Subroutine SumArray
L_Count,        DEC     0       //
L_Size,         DEC     0       // size of arrays
L_MinusSize,    DEC     0       // -size of arrays
L_ArrPtr1,      DEC     0       // temporary pointer to Array1
//
L_TmpStructPtr, HEX     0       //
L_Sum,          DEC     0       //
```

Note – the different style of commenting is for your understanding!

3. Write a subroutine **oddEven** that takes a number and determines whether it is odd or even, returning a 1 in the End-carry flag if is odd and 0 if it is even. The subroutine does not disturb the contents of the Accumulator on return.

Note: in order to ensure that the subroutine does not disturb the working-space (i.e. the Accumulator in this case), the subroutine has to store the value of the AC in a local (temporary) variable and retrieve it before returning from the subroutine.

Write the high-level algorithm before deciding how you are going to code your algorithm in assembly language. Use any transformations you think are necessary…

4. Implement the subroutine **findArrayMaxMin()** which finds the maximum value and the minimum value of a given array whose size is known. The values in the array are: $0 \leq value \leq +100$

(int min, int max) = findArrayMaxMin(int Array[], int Size);

Address of array

Write the high-level algorithm before you code the solution!

The **input** parameters (passed using the Stack) are:

- Array length = A_size.
- Array[] is the address of the start of the Array.

The **output** (return) parameters (passed using the Stack) are:

- The maximum value of the array.
- The minimum value of the array.

No parameter is passed through the Accumulator!

```
// Data local to main()
max,      DEC     0
min,      DEC     0
a_size,   DEC     4     //  array length
//
a_start,  HEX     100   //  pointer to Start of Array
          ORG     100
array,    DEC     1
          DEC     2
          DEC     3
          DEC     4

// Data local to subroutine
temp_max,     DEC   0
temp_min,     DEC   0
Count,        DEC   0          //  temporary iteration count
temp_size,    DEC   0
arrayPtr,     HEX   0          //  temporary array pointer
minus_1,      DEC   -1
```