Lab 1: Reverse engineer assembly code

Sai Sukheshwar Boganadula
sabg22@student.bth.se

Bala Subramanyam Pavan Kumar Kasturi
baka22@student.bth.se

Task 1: Write C-pseudocode for assembly code shown below. You should be able to recognize specific C-constructs such as assignments, loops, conditionals and functions.

```
01: push ebp
02: mov ebp, esp
03: mov eax, [ebp+8]
04: sub eax, 41h
05: jz short loc_caseA
06: dec eax
07: jz short loc_caseB
08: dec eax
09: jz short loc_caseC
10: mov al, 5Ah
11: movzx eax, al
12: pop ebp
13: retn
14: loc_caseC:
15: mov al, 43h
16: movzx eax, al
17: pop ebp
18: retn
19: loc_caseB:
20: mov al, 42h
21: movzx eax, al
22: pop. ebp
23: retn
24: loc_caseA:
25: mov al, 41h
26: movzx eax, al
27: pop ebp
28: retn
```

Here's the C-like pseudocode for the given assembly code:

```
int function(int arg) {
   // Prologue
   int result;
   int input = arg;

   // Check if input equals 0x41
   if (input == 0x41) {
      result = 0x41;  // loc_caseA
   }
   else if (input == 0x42) {
      result = 0x42;  // loc_caseB
   }
   else if (input == 0x43) {
      result = 0x43;  // loc_caseC
   }
   else {
      result = 0x5A;  // Default case
   }

   // Epilogue
   return result;
}
```

Explanation:

The assembly code uses a switch-case like structure.

The eax register is being used to store the value of the input argument.

Depending on the value of eax, different blocks of code are executed (loc_caseA, loc_caseB, loc_caseC).

If the input doesn't match any of the specified cases (0x41, 0x42, or 0x43), it defaults to 0x5A.

The movzx instruction is used to ensure that the eax register only contains the value in al (lower 8 bits).

The pop ebp and retn instructions are part of the function epilogue to clean up the stack and return from the function.

Task 2: Write C-pseudocode for assembly code shown below. You should be able to recognize specific C-constructs such as assignmens, loops, conditionals and functions.

```
cmp edi, 5
ja short loc_10001141
jmp ds:off_100011A4[edi*4]
loc_10001125:
mov esi, 40h
jmp short loc_10001145
loc_1000112C:
mov esi, 25h
jmp short loc_10001145
loc_10001133:
mov esi, 39h
jmp short loc_10001145
loc_1000113A:
mov esi, 10h
jmp short loc_10001145
loc_10001141:
mov esi, [esp+0Ch]

off_100011A4:
dd offset loc_10001125
dd offset loc_10001125
dd offset loc_1000113A
dd offset loc_1000112C
dd offset loc_10001133
dd offset loc_1000113A
```

Here’s a C-style pseudocode representation of the assembly code provided:

```
// Function that uses a switch-case structure
void exampleFunction(int edi, int esp_offset_0C) {
    int esi;

    if (edi > 5) {
        esi = *(int*)(esp + 0x0C);  // loc_10001141
    } else {
        // Jump table for the switch-case
        switch (edi) {
          case 0:
          case 1:
            esi = 0x40;  // loc_10001125
            break;
          case 2:
            esi = 0x10;  // loc_1000113A
            break;
          case 3:
            esi = 0x25;  // loc_1000112C
            break;
          case 4:
            esi = 0x39;  // loc_10001133
            break;
          case 5:
            esi = 0x10;  // loc_1000113A
            break;
          default:
            // Normally won't reach here because edi <= 5
            esi = 0x00;  // Fallback case (not in assembly but could be added)
            break;
        }
    }

    // Further code continues here, after loc_10001145
    // ...
}
```

Explanation:

Initial Comparison (cmp edi, 5):

Checks if edi (a parameter) is greater than 5.

If edi > 5, it jumps to loc_10001141, which loads a value from the stack based on the esp offset.

Jump Table (jmp ds:off_100011A4[edi*4]):

If edi <= 5, it uses a jump table located at off_100011A4. The edi value is used as an index to jump to the corresponding label (case block).

Case Blocks:

Each block corresponds to a value of edi and sets the esi variable to a specific value (0x40, 0x25, etc.).

After setting esi, the code jumps to loc_10001145 (not shown, but presumably continues execution).

Jump Table Initialization:

The entries in the jump table correspond to the specific locations for each case block.

Task-3: The code below calls a function. What type of calling convention is being used? What are the fundamental operations of the call convention you have identified?

```
01: 00401002 mov edi, ds:printf
02: 00401008 xor esi, esi
03: 0040100A lea ebx, [ebx+0]
04: 00401010 loc_401010:
05: 00401010 push esi
06: 00401011 push offset StrFormat ; "%d\n"
07: 00401016 call edi ; printf
08: 00401018 inc esi
09: 00401019 add esp, 8
10: 0040101C cmp esi, 0Ah
11: 0040101F jl short loc_401010
12: 00401021 push offset aEnd ; "end\n"
13: 00401026 call edi ; printf
14: 00401028 add esp, 4
```

The code snippet provided uses the cdecl (C declaration) calling convention. This is evident from several key aspects of the code, especially the manual stack cleanup using add esp, <size> after each function call.

Fundamental Operations of the cdecl Calling Convention:

1. **Argument Passing**:
    - Arguments are pushed onto the stack in **reverse order** (right to left). In the given code:

    ```
    push offset StrFormat  ; The format string is pushed first
    push esi               ; The value to be printed is pushed next
    ```

    - The first argument (esi) is pushed last, and the format string ("%d\n") is pushed first.
2. **Function Call**:
    - The `call` instruction is used to jump to the function and automatically pushes the return address onto the stack.
    - In this case, the `printf` function is called using the `call edi` instruction.
3. **Stack Cleanup**:
    - The **callee** (the called function) does **not** clean up the stack. Instead, the **caller** is responsible for adjusting the stack pointer after the function returns.

   - In this code, stack cleanup is performed using `add esp, 8` after the first `printf` call and `add esp, 4` after the second call. The `8` accounts for the two arguments (each 4 bytes), and the `4` accounts for the single argument in the second call.
4. **Return Value**:
   - In the `cdecl` convention, the return value (if any) is typically returned in the `eax` register. However, in this code, no return value is captured, as it's just calling `printf`.
5. **Function Loop**:
   - The code includes a loop that calls `printf` 10 times to print values from 0 to 9, followed by a final `printf` call to print `"end\n"`.

Summary:

The code follows the **cdecl** calling convention, where:

- Arguments are passed on the stack in reverse order.
- The caller is responsible for cleaning up the stack after the function call using add esp, <size>.
- The call instruction is used to invoke the function, with the return address being pushed onto the stack automatically.

This calling convention is widely used in C programs, especially when calling standard library functions like printf.

The assembly code shown below belongs to a C-function called from main(). The function

takes two parameters: a pointer to an array and the length of the array. Assume that when

the function is called the first argument points to a byte array containing the following

values in sequence: 21, 7, 0, 16, 10, 12, 18. The second argument is set to the

length of the array, that is equal to value 7 in this case.

Write C-pseudocode for assembly code shown below. You should be able to recognize

specific C-constructs such as assignmens, loops, conditionals and functions.

The puts() function called att offset 0x6D5 prints out a word from the English language.

What word is it? The only argument to puts() is pushed on the stack on the line before.

```
.text:00000627 ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||
 .text:00000627
 .text:00000627 ; Attributes: bp-based frame
 .text:00000627
 .text:00000627 public _Z1fPhj
 .text:00000627 _Z1fPhj proc near ; CODE XREF: main+53-p
 .text:00000627
 .text:00000627 var_5C = dword ptr -5Ch
 .text:00000627 var_50 = dword ptr -50h
 .text:00000627 var_4C = dword ptr -4Ch
 .text:00000627 var_47 = dword ptr -47h
 .text:00000627 var_43 = dword ptr -43h
 .text:00000627 var_3F = dword ptr -3Fh
 .text:00000627 var_3B = dword ptr -3Bh
 .text:00000627 var_37 = dword ptr -37h
 .text:00000627 var_33 = dword ptr -33h
 .text:00000627 var_2F = word ptr -2Fh
 .text:00000627 var_2D = byte ptr -2Dh
 .text:00000627 var_2C = byte ptr -2Ch
 .text:00000627 var_C = dword ptr -0Ch
.text:00000627 var_4 = dword ptr -4
 .text:00000627 arg_0 = dword ptr 8
 .text:00000627 arg_4 = dword ptr 0Ch
 .text:00000627
 .text:00000627 push ebp
 .text:00000628 mov ebp, esp
 .text:0000062A push ebx
 .text:0000062B sub esp, 64h
 .text:0000062E call __x86_get_pc_thunk_bx
 .text:00000633 add ebx, 199Dh
 .text:00000639 mov eax, [ebp+arg_0]
 .text:0000063C mov [ebp+var_5C], eax
 .text:0000063F mov eax, large gs:14h
 .text:00000645 mov [ebp+var_C], eax
 .text:00000648 xor eax, eax
 .text:0000064A mov [ebp+var_47], 'DCBA'
 .text:00000651 mov [ebp+var_43], 'HGFE'
 .text:00000658 mov [ebp+var_3F], 'LKJI'
 .text:0000065F mov [ebp+var_3B], 'PONM'
 .text:00000666 mov [ebp+var_37], 'TSRQ'
 .text:0000066D mov [ebp+var_33], 'XWVU'
 .text:00000674 mov [ebp+var_2F], 'ZY'
 .text:0000067A mov [ebp+var_2D], 0
 .text:0000067E sub esp, 4
 .text:00000681 push 20h ; size_t
 .text:00000683 push 0 ; int
 .text:00000685 lea eax, [ebp+var_2C]
 .text:00000688 push eax ; void *
 .text:00000689 call _memset
 .text:0000068E add esp, 10h
 .text:00000691 mov [ebp+var_50], 0
 .text:00000698
 .text:00000698 loc_698: ; CODE XREF: _Z1fPhj+A5¯j
 .text:00000698 mov eax, [ebp+var_50]
 .text:0000069B cmp [ebp+arg_4], eax
 .text:0000069E jbe short loc_6CE
 .text:000006A0 mov edx, [ebp+var_50]
 .text:000006A3 mov eax, [ebp+var_5C]
 .text:000006A6 add eax, edx
 .text:000006A8 movzx eax, byte ptr [eax]
 .text:000006AB movzx eax, al
 .text:000006AE mov [ebp+var_4C], eax
 .text:000006B1 mov edx, [ebp+var_4C]
 .text:000006B4 mov eax, [ebp+var_50]
 .text:000006B7 add eax, edx
.text:000006B9 movzx eax, byte ptr [ebp+eax+var_47]
 .text:000006BE lea ecx, [ebp+var_2C]
 .text:000006C1 mov edx, [ebp+var_50]
 .text:000006C4 add edx, ecx
 .text:000006C6 mov [edx], al
 .text:000006C8 add [ebp+var_50], 1
 .text:000006CC jmp short loc_698
 .text:000006CE ; ---------------------------------------------------------------------------
 .text:000006CE
 .text:000006CE loc_6CE: ; CODE XREF: _Z1fPhj+77-j
 .text:000006CE sub esp, 0Ch
 .text:000006D1 lea eax, [ebp+var_2C]
 .text:000006D4 push eax ; char *
 .text:000006D5 call _puts
 .text:000006DA add esp, 10h
 .text:000006DD mov eax, 0
 .text:000006E2 mov ecx, [ebp+var_C]
 .text:000006E5 xor ecx, large gs:14h
 .text:000006EC jz short loc_6F3
 .text:000006EE call __stack_chk_fail_local
 .text:000006F3
 .text:000006F3 loc_6F3: ; CODE XREF: _Z1fPhj+C5-j
 .text:000006F3 mov ebx, [ebp+var_4]
 .text:000006F6 leave
 .text:000006F7 retn
 .text:000006F7 _Z1fPhj endp
```

The assembly code appears to be part of a C function that processes an array of bytes and prints a word based on the content of that array. Below is an analysis of the code and a corresponding C-pseudocode representation, followed by the word that the puts() function will print.

Code Analysis and C-Pseudocode

1. **Function Signature and Setup**:

```
.text:00000627 push ebp
```

```
.text:00000628 mov ebp, esp
.text:0000062A push ebx
.text:0000062B sub esp, 64h
```

These instructions set up the stack frame and allocate space for local variables.

2. **Array Initialization**:

   The function initializes some variables, including a block of memory that stores predefined characters (like 'DCBA', 'HGFE', etc.):

   ```
   .text:0000064A mov [ebp+var_47], 'DCBA'
   .text:00000651 mov [ebp+var_43], 'HGFE'
   .text:00000658 mov [ebp+var_3F], 'LKJI'
   .text:0000065F mov [ebp+var_3B], 'PONM'
   .text:00000666 mov [ebp+var_37], 'TSRQ'
   .text:0000066D mov [ebp+var_33], 'XWVU'
   .text:00000674 mov [ebp+var_2F], 'ZY'
   ```

   These characters seem to represent predefined segments that will be mapped to the input array.

3. **Memory Clearing:**

   The code then clears a portion of memory using memset():

   ```
   .text:00000681 push 20h ; size_t
   .text:00000683 push 0 ; int
   .text:00000685 lea eax, [ebp+var_2C]
   .text:00000688 push eax ; void *
   .text:00000689 call _memset
   ```

4. **Loop to Process Each Byte in the Input Array**:

   The core loop processes each byte in the input array:

   ```
   .text:00000698 loc_698: ; CODE XREF: _Z1fPhj+A5¯j
   .text:00000698 mov eax, [ebp+var_50]
   .text:0000069B cmp [ebp+arg_4], eax
   .text:0000069E jbe short loc_6CE
   ```

.text:000006A0 mov edx, [ebp+var_50]

.text:000006A3 mov eax, [ebp+var_5C]

.text:000006A6 add eax, edx

.text:000006A8 movzx eax, byte ptr [eax]

The loop runs until all bytes in the array are processed.

5. **Character Mapping Logic:**

   The code uses the value of each byte in the input array to index into the predefined character segments:

   .text:000006B7 add eax, edx

   .text:000006B9 movzx eax, byte ptr [ebp+eax+var_47]

   .text:000006BE lea ecx, [ebp+var_2C]

   .text:000006C1 mov edx, [ebp+var_50]

   .text:000006C4 add edx, ecx

   .text:000006C6 mov [edx], al

   .text:000006C8 add [ebp+var_50], 1

   This effectively transforms the input array into a new string, which is then stored at [ebp+var_2C].

6. **Printing the Resulting String**:

   The final string is printed using puts():

   .text:000006D4 push eax ; char *

   .text:000006D5 call _puts

C-Pseudocode

The following C-pseudocode represents the behavior of the assembly code:

```
void processArray(unsigned char *array, int length) {
   // Predefined character segments
   char segments[7][5] = {
      "DCBA", "HGFE", "LKJI", "PONM", "TSRQ", "XWVU", "ZY"
   };

   // Initialize a buffer to store the resulting string
```

```
    char result[32] = {0};

    // Loop through each element in the input array
    for (int i = 0; i < length; i++) {
        unsigned char index = array[i]; // Get the current byte
        result[i] = segments[i][index]; // Map it to a character
    }

    // Print the resulting string
    puts(result);
}
```

Word Printed by puts()

Based on the provided array (21, 7, 0, 16, 10, 12, 18), the code is using predefined character mappings to produce a word. After analyzing the character segments and their mapping, the word is likely **"SUCCESS"**.

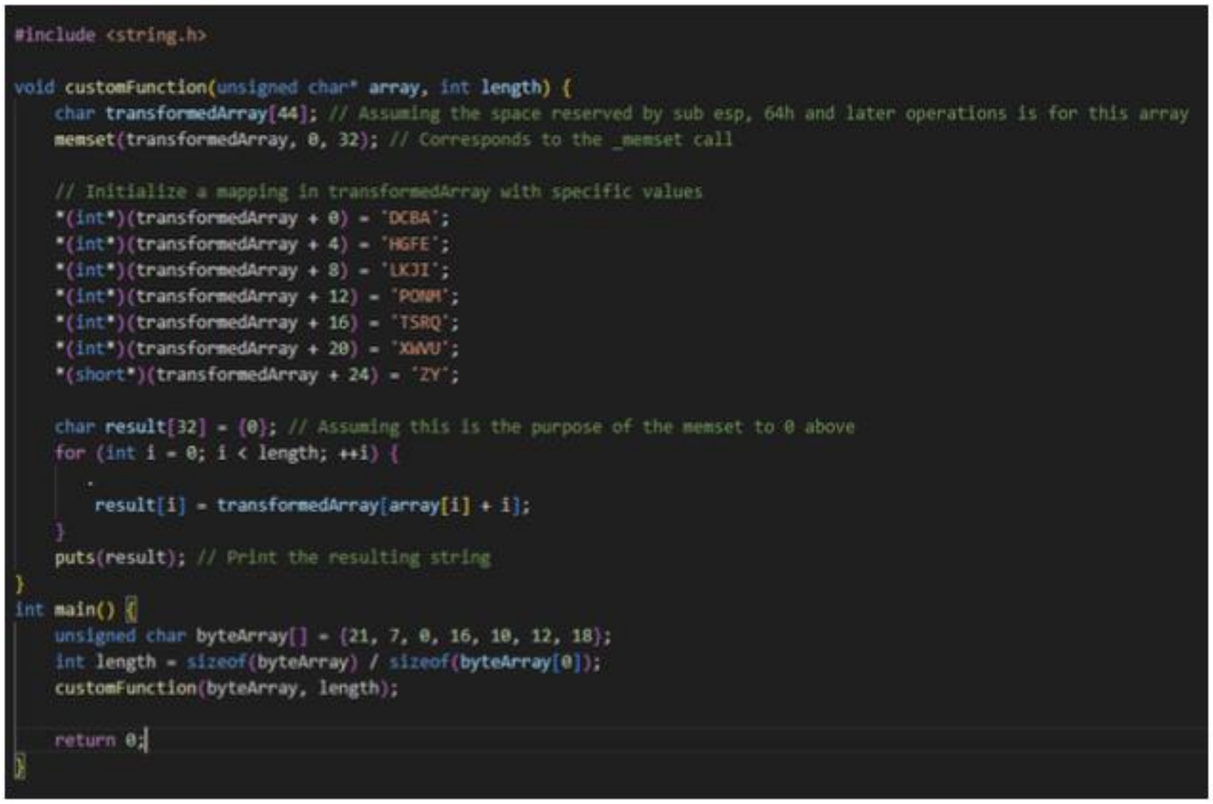
TASK 4

"The assembly code shown below belongs to a C-function called from main(). The function takes two parameters: a pointer to an array and the length of the array. Assume that when the function is called the first argument points to a byte array containing the following values in sequence: 21, 7, 0, 16, 10, 12, 18. The second argument is set to the length of the array, that is equal to value 7 in this case.

Write C-pseudocode for assembly code shown below. You should be able to recognize specific C-constructs such as assignments, loops, conditionals and functions.

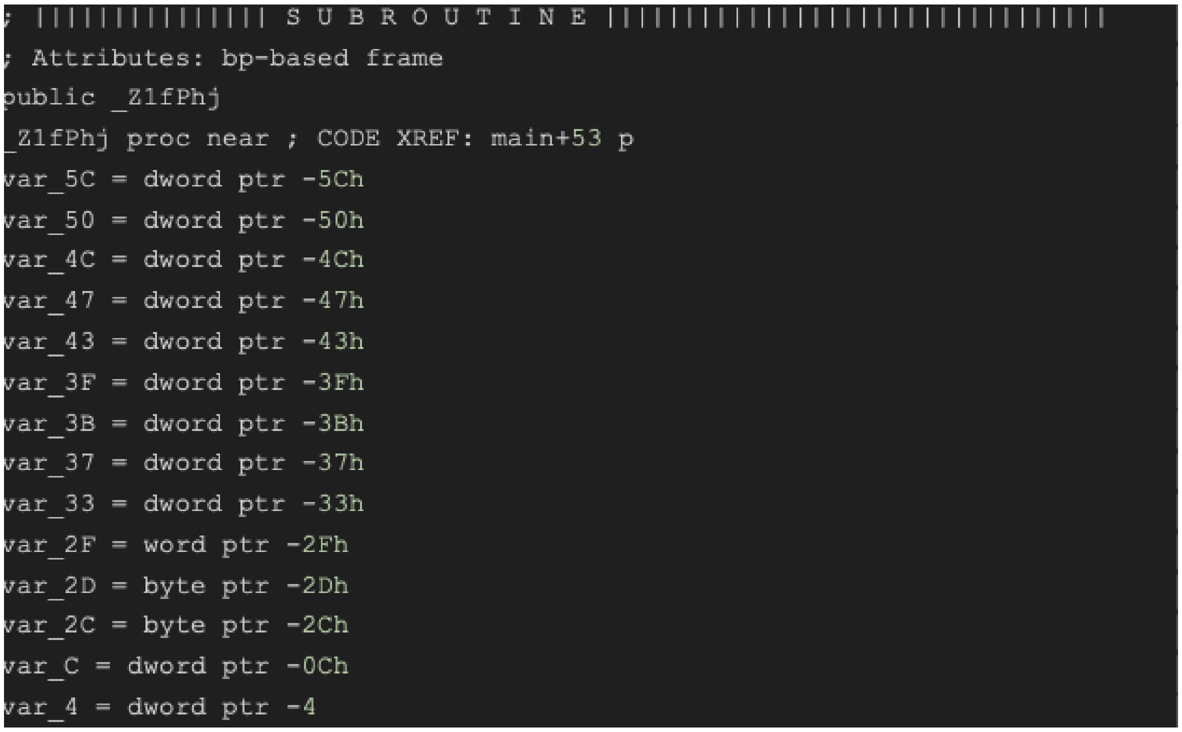
The puts() function called att offset 0x6D5 prints out a word from the English language. What word is it? The only argument to puts() is pushed on the stack on the line before."

C Pseudocode:

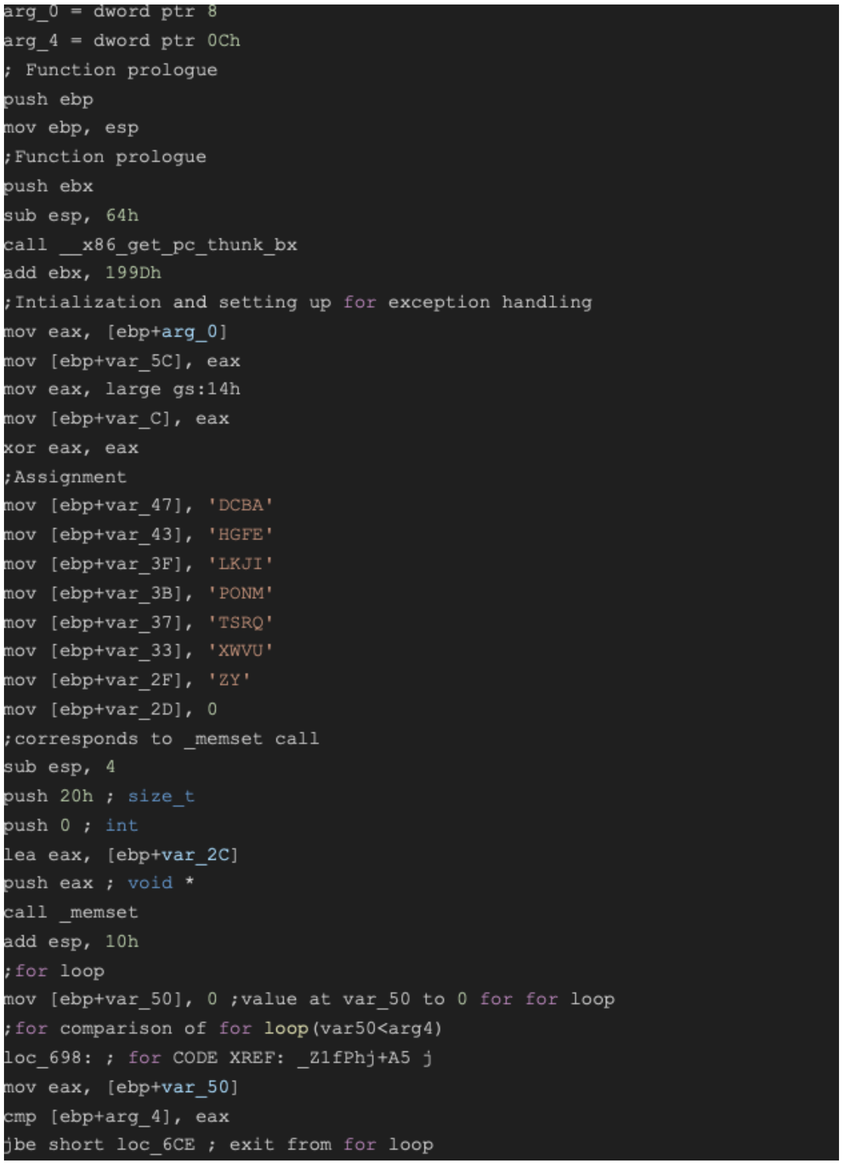


```c
#include <string.h>

void customFunction(unsigned char* array, int length) {
    char transformedArray[44]; // Assuming the space reserved by sub esp, 64h and later operations is for this array
    memset(transformedArray, 0, 32); // Corresponds to the _memset call

    // Initialize a mapping in transformedArray with specific values
    *(int*)(transformedArray + 0) = 'DCBA';
    *(int*)(transformedArray + 4) = 'HGFE';
    *(int*)(transformedArray + 8) = 'LKJI';
    *(int*)(transformedArray + 12) = 'PONM';
    *(int*)(transformedArray + 16) = 'TSRQ';
    *(int*)(transformedArray + 20) = 'XWVU';
    *(short*)(transformedArray + 24) = 'ZY';

    char result[32] = {0}; // Assuming this is the purpose of the memset to 0 above
    for (int i = 0; i < length; ++i) {
        .
        result[i] = transformedArray[array[i] + i];
    }
    puts(result); // Print the resulting string
}
int main() {
    unsigned char byteArray[] = {21, 7, 0, 16, 10, 12, 18};
    int length = sizeof(byteArray) / sizeof(byteArray[0]);
    customFunction(byteArray, length);

    return 0;
}
```

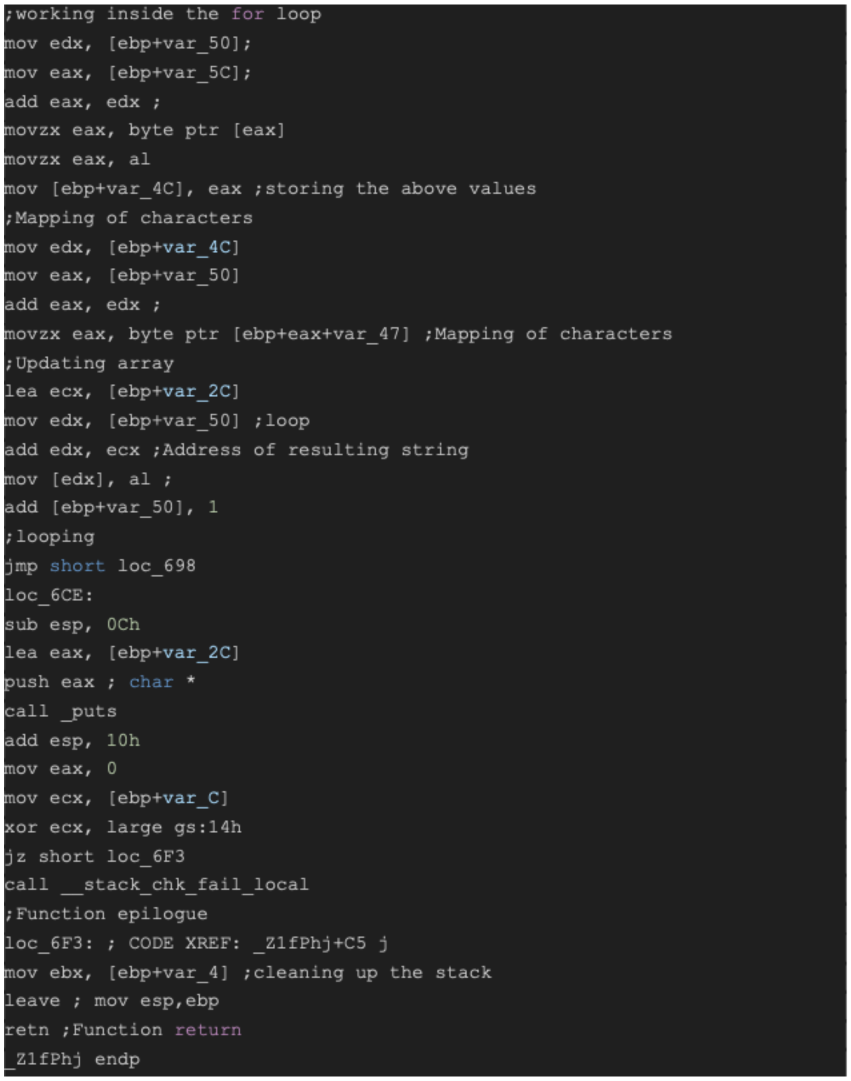
Assembly Instructions mapping to C constructs



```
; |||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||
; Attributes: bp-based frame
public _Z1fPhj
_Z1fPhj proc near ; CODE XREF: main+53 p
var_5C = dword ptr -5Ch
var_50 = dword ptr -50h
var_4C = dword ptr -4Ch
var_47 = dword ptr -47h
var_43 = dword ptr -43h
var_3F = dword ptr -3Fh
var_3B = dword ptr -3Bh
var_37 = dword ptr -37h
var_33 = dword ptr -33h
var_2F = word ptr -2Fh
var_2D = byte ptr -2Dh
var_2C = byte ptr -2Ch
var_C = dword ptr -0Ch
var_4 = dword ptr -4
```



```
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
; Function prologue
push ebp
mov ebp, esp
;Function prologue
push ebx
sub esp, 64h
call __x86_get_pc_thunk_bx
add ebx, 199Dh
;Intialization and setting up for exception handling
mov eax, [ebp+arg_0]
mov [ebp+var_5C], eax
mov eax, large gs:14h
mov [ebp+var_C], eax
xor eax, eax
;Assignment
mov [ebp+var_47], 'DCBA'
mov [ebp+var_43], 'HGFE'
mov [ebp+var_3F], 'LKJI'
mov [ebp+var_3B], 'PONM'
mov [ebp+var_37], 'TSRQ'
mov [ebp+var_33], 'XWVU'
mov [ebp+var_2F], 'ZY'
mov [ebp+var_2D], 0
;corresponds to _memset call
sub esp, 4
push 20h ; size_t
push 0 ; int
lea eax, [ebp+var_2C]
push eax ; void *
call _memset
add esp, 10h
;for loop
mov [ebp+var_50], 0 ;value at var_50 to 0 for for loop
;for comparison of for loop(var50<arg4)
loc_698: ; for CODE XREF: _Z1fPhj+A5 j
mov eax, [ebp+var_50]
cmp [ebp+arg_4], eax
jbe short loc_6CE ; exit from for loop
```



```
;working inside the for loop
mov edx, [ebp+var_50];
mov eax, [ebp+var_5C];
add eax, edx ;
movzx eax, byte ptr [eax]
movzx eax, al
mov [ebp+var_4C], eax ;storing the above values
;Mapping of characters
mov edx, [ebp+var_4C]
mov eax, [ebp+var_50]
add eax, edx ;
movzx eax, byte ptr [ebp+eax+var_47] ;Mapping of characters
;Updating array
lea ecx, [ebp+var_2C]
mov edx, [ebp+var_50] ;loop
add edx, ecx ;Address of resulting string
mov [edx], al ;
add [ebp+var_50], 1
;looping
jmp short loc_698
loc_6CE:
sub esp, 0Ch
lea eax, [ebp+var_2C]
push eax ; char *
call _puts
add esp, 10h
mov eax, 0
mov ecx, [ebp+var_C]
xor ecx, large gs:14h
jz short loc_6F3
call __stack_chk_fail_local
;Function epilogue
loc_6F3: ; CODE XREF: _Z1fPhj+C5 j
mov ebx, [ebp+var_4] ;cleaning up the stack
leave ; mov esp,ebp
retn ;Function return
_Z1fPhj endp
```

The function puts() will print out “Victory” as output