In this lab, you are to decode and display a message that has been compressed using Huffman coding. Each character in the message is represented by a unique substring of bits. The code is optimized so that more common characters are represented using fewer bits than less common characters. The substrings are concatenated to form one long string. For example, the word “Mississippi” could be represented as 100110011001110110111, where substrings translate into letters according to the table on the right.

In facilitate decoding, the table is converted into a binary tree stored as an array of bytes. The byte representing a leaf node contains an ASCII character, while the bytes of all interior nodes contain the integer 0. The root node’s byte is stored at array index 0. In general, if the byte index of a node is \( k \), the byte index of its left child is \( 2^{k} + 1 \) and that of its right child is \( 2^{k} + 2 \).

The coded message is decoded and printed using the following algorithm:

```
start:   k ← 0; // Start at the root of the tree
top:     bit ← GetBit; // Get the next bit of the coded message
         k ← 2 \times k + 1 + bit; // If bit = 0 descend left, else descend right
         byte ← array[k]; // Get the content of the node
         if byte = 0 goto top; // If it’s zero, it’s an interior node
         if byte = '$' return; // If it’s a dollar sign, you’re done
         Display1Char(byte); // Otherwise it’s a leaf node: print the character
         goto start; // Go back to top of tree and decode next character
```

Implement this algorithm as a function written in assembly. The function prototype has two parameters: \( \text{msg} \) is a pointer to the bits of the coded message packed 8 per byte and \( \text{array} \) holds the binary decoding tree. The first bit of the message is the least-significant bit of the first byte of memory pointed to by parameter \( \text{msg} \).

```
void DecodeMessage(void *msg, char array[]);
```

You will also need to implement a second function in assembly. Function \( \text{GetBit} \) returns either 0 or 1, corresponding to a bit of the coded message located at position \( \text{bitnum} \) (starting at position 0).

```
int GetBit(void *msg, uint32_t bitnum);
```

Implement \( \text{GetBit} \) two ways: (1) use bitwise and shift operations, and (2) use Bit-Banding.

Function \( \text{Display1Char} \) is implemented in the main program. Test your solution using the C main program downloaded here.