Note: all numbers are in decimal unless specified otherwise

Factorial is achieved by multiplying the input (which will be called n for the rest of this) by n-1, n-2 … n-(n+1). In a higher level programming language, this can be achieved by using recursion to multiply n by n-1… until n hits 1 (see C code below). In assembly, we will use 16 bit addition (see below for an explanation on how to do this with only 8 bit memory and registers) to multiply the numbers in order to get the factorial. Lastly, we will need to convert from the binary representation inside of the two 8 bit registers to Hexadecimal by using nibbles and addition (see below for a full explanation).

```
int factorial(input) {
    if(input == 0){
        return 1;
    }
    return n*factorial( input: input-1);
}
```

Since assembly has no while or for loop function built in like higher level programs such as C, in order to repeat commands an unknown number of times, we have to do a couple things. First, we use a counter variable (declared after the HLT command) and decrement it by adding another variable with a value of -1 (decimal). After that, we check if the counter variable has gotten to 0 yet by loading the counter variable (though the LDA command) and using SZA (Skip next instruction if AC is 0) to skip the “looping” instruction if the counter is 0. The instruction used to “loop” is BUN (branch unconditionally) which can be given the address of the beginning of the “loop” to repeat the “loop” until the counter variable is 0, where it will be skipped by SZA and the program will continue on with the next instruction (see image).

```
TEST, ADD D
OUT
BUN TEST
```

Another challenge for this program is the need to add and multiply 16 bit numbers, when the registers and memory are only 8 bits. In order to add 16 bit numbers, we will use two 8 bit registers (for our example, we will call the two registers for the first number being added A and B and the two registers for the second number being added X and Y). We will first clear the E bit and then add B and Y (the least significant registers). After that, we will save B+Y in register B before clearing AC, rotating once to the left (to get the carry "E" bit in the least significant bit slot) before adding the rotated E bit with the A and X registers.

```
LDA B
ADD Y
STA B
CLA
CIL
ADD A
ADD X
```

In order to calculate factorial, we will be using several "loops". One loop (which will be called LOOP inside of my program) will be used to count from the input number down to 1 in order to calculate the factorial. The second "loop" (called MULTIPLY) will be used for multiplication of n with n-1.

LOOP will first be set to the input number, and will slowly be decremented by adding -1 until it reaches 0, where the "loop" will be exited by skipping the BUN call.

The counter within MULTIPLY will be set at first to the LOOP counter and will be used to add (using the method above for adding 16 bit numbers) the LOOP counter number a certain number of times (for example, if the LOOP counter is 5, 5 will be added 6 times in order to achieve multiplication 6*5).

Inside of the computer, all of the numbers are represented in binary, not in decimal. This means that we do not have to worry about converting from decimal to hex, but we only have to worry about converting from binary to hex. In order to convert from binary to hex

(with 8 bit registers), we first need to separate the 8 bit registers into a 4 bit “nibble” (which will be stored in an 8 bit “variable” with values 0000XXXX, with the Xs representing the 4 bits). For the 4 least significant bits in each 8 bit register (we have a 16 bit number, which is represented by 2 8 bit registers), it is easy to get the nibble, as you only need to use the AND operator with hexadecimal 0F (this will set the most significant 4 bits to 0 and and the least significant 4 bits to the values in the original register).

For the most significant 4 bits in the least significant register (we do not care about the most significant 4 bits in the more significant register as they are guaranteed to be 0 by the input limits), it is a bit harder to convert them into a nibble. In order to do this, we are going to clear the E bit using the CLE command, and then use the AND operator with value F0 (hexadecimal) to only be left with the rightmost bits. After that, we circular shift the register to the right 4 times in order to get a nibble.

The last step is to detect if the nibble is less than 10 or greater than/equal to 10. If it is less than 10, we add it with 30 (hex) using the ADD command, and if it is greater than or equal 10, we add it to 41-A (both numbers are hex) using the ADD command as well. We then print out the converted character and then move on to converting the other remaining bits (a total of 3 nibbles need to be converted into hexadecimal characters).