****

**Universidad Autónoma de Guadalajara**

Ingeniería Electrónica Biomédica

Arquitecture of Microcontrolers

Andrea Alejandra Mondragón Olivos

Jesús Arnoldo Zerecero Núñez

2915351

2885993

*“Practice 2”*

28-06-17

Functions in assembler

**Introduction.**

Assembler is a low level programming language which can be used to generate simple step by step instructions for a program to follow. Because of its simple step by step structure, computers are able to understand it directly, with close to no decoding whatsoever. This language utilizes a limited amount of registers to carry data from one place to another.

Programs written in assembler are written by joining simple instructions, one at a time, indicating when to access memory, loading from it or storing in it, as well as designating which registers to use in each instruction. Higher level programming languages are able to complete multiple actions or instructions at once. This instructions also know when to access memory which makes codes simpler and shorter, but requires more processing and a longer decoding.

There are certain functions high level languages which already know the set of instructions to execute in order to carry out the desired result. The goal of this practice is to recreate three of this instructions in assembler language, coding one by one every instruction needed in the simplest way possible:

* **MemCopy**: This function reads an array of data and copies it in a different memory space, loading and storing between arrays one element at a time It receives two memory pointers to arrays and has no return parameters.
* **Strcmp**: This function reads two strings of characters, element by element and compares them to see if they are identical or not. It receives two memory pointers to strings and returns a “0” if the strings are identical or a “1” if they are different.
* **Strlen**: This function reads a single string of characters and counts how many letters or elements are in it, excluding the null character. It receives a memory pointer to a string and returns a numeric value defined by an accumulator which increases every time a character different from null is counted.

**Development.**

1. Create a separate CodeWarrior project for each of the three functions, using assembler language:

**MemCpy:**

1. Define a macro which holds the number size of both arrays.
2. Declare two variables to hold the addresses of both arrays (source and destination) and fill them with any numbers.
3. In the main function, load the arrays’ addresses to two registers, and load the element size to a third register. This are the parameters the function will be receiving.
4. Make a call to the function and declare it after main.
5. The function will go through both arrays at the same time, loading the current source array value into a register and storing it into the current destination array position.
   * This position is defined by a register offset value used in both loading and storing.
   * This register will be increased by one at the end of each cycle
6. Make a comparison between the offset register and the register holding the element size.
   * The cycle will repeat until this comparison yields an equal, as this means the last array element has been passed on.

**Strcmp**

1. Declare two variables to hold the addresses of both strings to compare and initialize the strings with any word.
2. In the main function, load the strings’ addresses to two registers.
3. Make a call to the function and declare it after main.
4. In the function, the letters of both strings in the current position must be loaded into two new registers in order to be compared.
   * The position is also defined by a register offset value, which will be increased by one at the end of each cycle.
5. Firstly, both registers holding the letter must be compared with zero in order to know if the end of one of the strings was reached at that position.
   * In case a null value was found (the comparison with zero yields equal), the cycle ends, and the letters will be compared one last time in order to decide if both nulls were reached at the same time, which would mean both strings are identical in length.
     1. If both nulls were reached, a last branch is called where r0 is assigned a value of “0”. This means the result of the function returns a “0” value.
     2. If only a null was reached, the code branches to a different place, where r0 is assigned a value of “1”, which means the strings are not identical.
6. In case none of the letters were null at that time, a comparison between them is made.
   * If they are different, the code branches to where r0 is assigned a value of “1”.
   * If they are equal, the index increases and the cycle is repeated.

**Strlen**

1. Declare a variable to hold the address of the string to measure, and initialize the string.
2. In the main function, load the string address to a register.
3. Make a call to the function and declare it after main.
4. Initialize the offset and accumulator (r0 as return parameter) register in zero.
5. In the function, the current letter of the current position must be loaded into a different register.
   1. The position is also defined by the register offset value, which will be increased by one at the end of each cycle. The accumulator register is increased by one as well.
6. Compare the current letter with zero, in order to know if the end of the string was reached.
   1. If a null was found, the cycle ends and r0 is subtracted by one to discount the null as a letter in the accumulator.
   2. If a null wasn’t found, the cycle repeats.

This are very simple codes in our opinion, as programming them wasn’t a harsh task to accomplish. All codes were written in less than an hour each. The harder parts of this practice were taking care of the format to write the code (labels, using appropriate lower and upper cases, proper spacing and proper variable and function naming). Also, we had to research the correct way to declare string type variables, as we had never done that before in MCU class. We spent a lot of time in this research, as we weren’t able to find much information about assembler code in CodeWarrior on the internet. The most time consuming task was researching how all three functions work step by step. We based our research on how they work in C language so we could recreate the functions accordingly in the simplest way possible in assembler language.

Another major complication we encountered was not being able to update our KL25 MCU, as the operating system of both our laptops was windows 8.1 or higher, so we had to ask a friend to update our MCU using his laptop.

The tests performed while debugging all three functions were successful, carrying out multiple tests and examining all corner cases before approving the performance of our codes.

**Conclusion.**

**Arnoldo:**

In conclusion, coding in assembler language allows you to have a much bigger understanding of the program’s functioning. It shows how computers read a high-level programming code after decoding it. It is much more time consuming for the programmer to write, but it makes it easier for readers to understand what the code is essentially doing, as all steps are shown, without comprising them in functions. Personally, I enjoy programming in this language, as I feel I’m learning much more about computers at an internal decoding level. I just wouldn’t like using this language to program longer, more complex codes which require multiple functions which could be reduced in a single line of code in other languages.

**Alejandra:**

I can conclude that assembler language helps us comprehend the internal processes step by step of data processing inside a microcontroller, as well as in the different computing systems for data managing. Assembler is a very descriptive language as it shows you what the code does step by step all the way to the result, using commands and registers pre-programmed in the microcontroller.