

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | Conversion of a 4-digit unsigned packed BCD number to a16-bit binary format in Assembly programming language. | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Student’s Full Name: Türker ErbasiNeptun Code: NVTRE |  | | | |
|  | | | | Microcontroller Based Systems BMEVIAUAC06—Lecturers:Gábor Tevesz Dr., Domokos Kiss, Viktor Kovács—2021/22/1 |  | | | |
|  | | |  | | |  | | |

##### Table of Contents

1. [Table of contents](#_Table_of_Contents)
2. [Declaration](#_Declaration)
3. [Description of the program](#_Description_of_the)
4. [Logic behind the program](#_Logic_Behind_the)
5. [Description of the subroutines](#_Descriptions_of_the)
   * 1. [BCD2BIN](#_BCD2BIN)
     2. [ISOLATE\_NIBBLES](#_ISOLATE_NIBBLES)
     3. [MUL10](#_MUL10)
6. [References](#_References)

##### Declaration

I hereby declare that the documentation of my project and my project has been composed by myself alone.

##### Description of the program

The packed binary coded decimal is a numerical format that relies on the representation of a decimal digit with four bits which is more than enough to represent a decimal digit.

Diagram, schematic

Description automatically generatedIn this task I did conversion between the numerical format mentioned above and the 16-bit binary format. In order to complete this task in Assembly programming language I used some subroutines and applied some well-known techniques.

##### Logic Behind the Program

The basic principle of the program is first to separate the packed BCDs into two bytes (done in main), then isolate the nibbles of the bytes (ISOLATE\_NIBBLES) and create the four 4-bit binary numbers of each digit of the packed BCD. Temporary storage of these digits is R2, R3, R4, R5 registers, respectively. After the separation is complete multiplication subroutine (MUL10) multiplies the more significant nibble by 10 and adds the less significant nibble of the higher two 4-bit binary numbers. The result is an 8-bit binary number. When it returns to the conversion subroutine (BCD2BIN) it multiplies the result by 100 in order to prepare it for the final sum. Then it does the same multiplication (MUL10) for the lower half of the packed BCD. Finally, to get our converted 16-bit binary number it sums them up while taking into consideration the carry bit and putting the values to the high and the low halves of the 16-bit binary number to the R4 and R5 registers respectively. An example with the given number in the project template is shown in Figure 1.

Figure 1: Basic Principle of the Program with an example packed BCD number from the template project

##### Descriptions of the Subroutines

##### BCD2BIN

**Inputs**: R6 - Higher 2 BCD digits, R7 - Lower 2 BCD digits

**Outputs**: R4 - High byte of the parsed 16-bit number, R5 - Low byte of the parsed 16-bit number

**Modifies**: Accumulator, B-register, R4, R5, R6, R7 registers.

The working principle of the subroutine is: In order to point other registers address of the R2 register moved to the R0 register. In order to separate the nibbles of the BCD, it loads the Accumulator **with the higher two BCDs** and calls the **“ISOLATE\_NIBBLES”** subroutine. When this subroutine returns, it loads the Accumulator with **the lower two BCDs** and calls the **“ISOLATE\_NIBBLES”** subroutine again. Until here we already have all BCDs separated and stored at the R2, R3, R4, R5 registers. Then moves address of the R2 registerto the R0 register to point again and calls the **“MUL10”** subroutine. This is because we want to multiply the more significant nibbles in each high and low bytes of the BCD. After the **“Mull10”** subroutine, since these two digits are thousands and hundreds digits, we multiply it with decimal 100 (for example: 0011 1011b = 59d by multiplying, it becomes 0001 0111 0000 1100b = 5900d). Moves the higher and lower bytes of the result to the R6 and R7 registers, respectively, for storing. Then repeats the **“MUL10”** subroutine for **the lower two BCDs**, but this time it does not multiply by 100 since these two digits are tens and ones digit.

At last, it moves back the lower byte of **the higher two BCDs** to the B-register and adds them together with the current value of Accumulator which is the result of the last **“MUL10”** subroutine. Moves the result to the R5 register. In case of a carry, R4, the higher byte of **the higher two BCDs** moved to the Accumulator and by using “ADDC” instruction sums up with zero in order to get the carry bit if exists. Then it moves the result back to R4 and returns.



##### ISOLATE\_NIBBLES

Diagram

Description automatically generated**Input**: Accumulator - Input number high/low byte of a BCD number

**Outputs**: @R0(R2, R3, R4, R5) - Output number isolated 4-bit (nibble) binary numbers of the BCD

**Modifies**: Accumulator, B-register, R2, R3, R4, R5 registers

This subroutine aims to separate each nibble of the BCD. The technique applied in this subroutine is called masking. With the “ANL” instruction, which is a bitwise logical “AND” operation among the specified bit or word operands. The Masking technique is used as a set or a reset of the binary numbers. The programs use the “AND” operation for resetting the more significant bits we do not want to have. For example, by using “ANL” instruction with 0101 1001b and 0000 1111b operands, it achieves to store 0000 1001b binary number in the Accumulator since the instruction “masked” (“reset”) the “0101” part of the binary number.

The “ISOLATE\_NIBBLES” subroutine, has input as a byte form of the two higher or the lower BCDs. After moving the input, also into the B-register, it “masks” the more significant 4-bit part of the byte in the Accumulator by 1111 0000b, and by using “SWAP” instruction it switches the low-order and high-order nibbles within the accumulator to have the first digit of the BCDs. Then do the same “ANL” instruction with the B-register and 0000 1111b, but this time it does not use “SWAP” instruction since it is already in the less significant side of the byte. It moves the values of the Accumulator and the B-register to the address at R0, which is, in this case R2 and R3 registers, respectively. After these instructions, it returns. In case of the lower BCDs only difference is that it moves the results to the address at R0, which is R4 and R5 registers, respectively. Therefore, it has all results in the data memory in separated forms after the second “CALL” of the subroutine.

Figure 2: Flowchart of the program

##### MUL10

**Input**: @R0 - Input 4-bit binary number of higher/lower nibbles of the higher/lower byte

**Outputs**: A - Output 8-bit binary number sum of the higher multiplied by 10 and lower nibbles

**Modifies**: Accumulator, B-register, R0 register

“MUL10” subroutine is the simplest amongst others. Its purpose is simple and does not use complex instructions. When first called, it starts the subroutine by moving a decimal 10 value to the B-register and the value which is pointed by the R0 register which is pointing at the address of the R2 register currently since in the previous subroutine (BCD2BIN) moved the “#02h” value again before calling this subroutine. With the “MUL” instruction it gets the multiplication of them. We can think of the reason for the multiplication as shifting a digit of a decimal number. After getting the result, it increments the value in the R0 register in order to get the next value. At the end sums the result of the multiplication with the value at R0 pointing at end returns.



##### References

* Sideshows presented at the lectures.
* Datasheet available at the faculty’s website.
* <https://www.electronicshub.org/8051-microcontroller-instruction-set/>
* <https://www.keil.com/support/man/docs/is51/>
* <https://www.aplawrence.com/Basics/packedbcd.html>
* <https://en.wikipedia.org/wiki/Mask_(computing)>
* <https://www.tutorialspoint.com/embedded_systems/es_registers.htm>