European Funds
Knowledge Education Development

European Union
European Social Fund

Faculty of Automatic Control, Electronics and Computer Science

Informatics, 1st cycle of higher education, semester 1

Programming in assembler language

Programming in assembler language is not much different from programming in high-level languages. The main difference is the need to use a much poorer set of instructions. In assembler languages, for example, there are no structural instructions or instructions that allow you to easily describe the calculation of an arithmetic expression. However, anything you can write in high-level languages can certainly be expressed using assembler instructions. You can also do many things that are impossible to easily formulate in high-level languages. A programmer writing in assembler language has full control over what is happening during calculations in the processor. It is also not limited by various rules in high-level languages. However, full control and no restrictions mean also full responsibility for the written program.

When writing a program in assembler language, we use almost exclusively the instructions corresponding to the instructions of a given processor. Therefore, it is necessary to know these instructions, as well as familiarize yourself with the architecture of the processor. The assembler language program consists of instructions and data on which these instructions operate. The program is a sequence of successive lines, in each line there may be one instruction statement or a single data declaration. Formally, the syntax of the program line is as follows:

[<label>:] <statement> [<argument>]

where:

- <label> – a string of letters and numbers that is a symbolic representation of the specified address
- <statement> – symbolic name (mnemonics) of one of the processor's instructions or one of the so-called pseudo-instruction (declaration of space in memory for data: RST, RPA)
- <argument> – a decimal number or one of the labels entered at the beginning of the lines

The RST and RPA pseudo instructions allow you to appropriately reserve a memory location for a single data with the initial value (RST) given as an argument and to reserve a place in memory for the data without indicating its initial value (RPA). As the instruction statement, the name of one of the available instructions may appear. We will assume that there are 8 instructions in the machine's W processor listed in the table 1.

Table 1. Instruction list

Mnemonic	Opcode	Operation
STP	000	Stop (end) the execution of the program
DOD	001	Adding to the accumulator AK the contents of the memory location indicated by the argument
ODE	010	subtracting from the accumulator the contents of the memory location indicated by the argument
POB	011	retrieving the contents of the memory location indicated by the argument to the accumulator
ŁAD	100	loading the contents of the accumulator to the memory location indicated by the argument
SOB	101	indication that the next instruction to execute will be the one that is in the memory location indicated by the argument (unconditional jump)
SOM	110	If there is a negative number in the accumulator, the next instruction will be executed in the memory location indicated by the argument. If there is a non-negative number in the accumulator, the next instruction will be carried out in memory directly after the SOM instruction
SOZ	111	The jump to the address indicated by the argument is executed only when the accumulator is 0. Otherwise the instruction placed in the memory immediately after the SOZ instruction will be performed as the next one.

To write a program in assembler language you must first create an algorithm that solves a specific task, specify it using only available instructions and finally save it in the form of a program in assembler language. Writing these types of programs will be illustrated by several examples.

Example 1

Write a program calculating the product of two natural numbers *a* and *b* and placing the result in the variable labeled *c*.

The first step in solving this task is to come up with the right algorithm. To find the product of two natural numbers a and b, it is enough at the very beginning to assume that it is equal to 0, and then a ties increase its value by b. This is illustrated in the block diagram in the figure 1

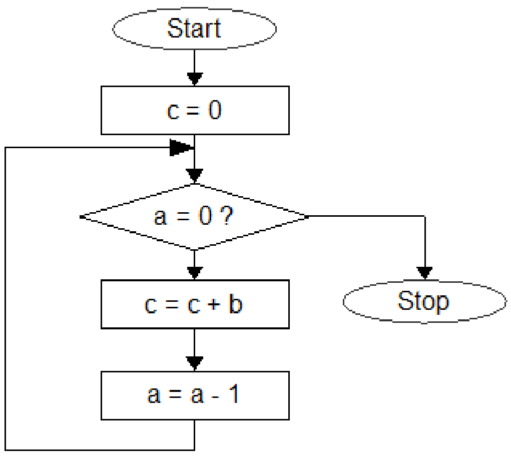

Figure 1. Multiplication algorithm

Below is a listing of the appropriate program in a situation where the variables a and b are equal to 4 and 5 respectively. Each operation on the diagram (figure 1) will usually correspond to one or more lines in the assembly language. Note that assigning the initial value of the variable c is not represented by the sequence of machine instructions but it was implemented by using the pseudo-instruction RST (line 14 in the code). Testing whether a takes a value of 0 implements the code placed on lines 1 and 2. If the condition is met (a = 0), terminate the loop, retrieve the result to the accumulator and terminate the program (lines 10 and 11).

```
1.          POB a
2. loop:    SOZ end
3.          POB c
4.          DOD b
5.          ŁAD c
6.          POB a
7.          ODE One
8.          ŁAD a
9.          SOB loop
10. end:    POB c
11.         STP
12. a:      RST 4
13. b:      RST 5
14. c:      RST 0
15. One:    RST 1
```

Increasing the value of the variable *c* by the value of *b* is accomplished by the statements from lines 3, 4 and 5, while the decrementation of the variable *a* is in lines 6, 7 and 8. Note that the argument to the subtraction command (line 7) is not the number 1 (ODE 1) but the *One* label pointing to a memory location containing one value (line 15). In our processor, the direct addressing mode is used, which means that the ODE 1 statement would be interpreted as a desire to subtract from the accumulator the contents of the memory location with address 1, meanwhile at this address the instruction in the second line of the program's code will be placed in memory. Finally, in line 9, the program returns to the beginning of the loop to test its termination condition.

The given program can be assembled (translated into machine code) and, after loading into the main memory, executed. After its execution, the accumulator should have the final value of the variable marked with the label c, in this case the value of 20.

Array processing

The instructions listed in Table 1 allow you to express many even very complex programs operating on a variety of data. However, manipulating them with data that make up complex structures is not a simple task. This requires the implementation of quite complicated transformations that use knowledge about the location and organization of data in the computer's memory. This problem will be illustrated by an example.

The computer's memory has an *n*-element array. The first element of this array is stored in the memory location marked with the *Array* label. For the memory location labeled *Sum*, enter the sum of all *n* elements in the array.

Before we present a solution to the task, let's explain how the elements of the array are arranged in memory. The array in the computer memory creates a coherent area, each subsequent element of the array is saved in the next location of this area. For example, a four-element array filled with four consecutive natural numbers will be written in assembler language as follows:

```
Array:  RST 1
        RST 2
        RST 3
        RST 4
```

Knowing the address of the first element of an array we can easily determine the location in the memory (address) of any other element of the array. We will assume that the array, like in C, will be indexed from 0. To enumerate the address in the memory of the third element of the array (that is, element with index 2), one should add the index of the corresponding element to the address of the beginning of the array. If the array starts with a memory location with address 20, the third element of this array can be found at address 22 (20 + 2 = 22).

To sum up all elements of an array, we will start by resetting the `Sum` variable. Next, we will add to the variable `Sum` in the loop the values of subsequent elements starting from the first one, that is, the one which is located under the symbolic address represented by the label `Array`. In each successive run of the loop, the value of the next element of the `Array` will be added, i.e. the address of the added element after each loop run, we need to increase by 1. This type of task would be easily performed using instructions with indirect or index addressing. Many processors offer such instructions. In the instruction list of our computer, however, we will not find instructions using other addressing modes than direct addressing. Therefore, our solution will require modifying the content of the program during its execution.

```
1. loop:   POB n
2.         ODE One
3.         SOM end
4.         ŁAD n
5.         POB Sum
6. Instr:  DOD Array
7.         ŁAD Sum
8.         POB Instr
9.         DOD One
10.        ŁAD Instr
11.        SOB loop
12. end:   POB Suma
13.        STP
14. n:     RST 4
15. Array: RST 1
16.        RST 2
17.        RST 3
18.        RST 4
19. Sum:   RST 0
20. One:   RST 1
```

After each loop run, you must modify the contents of the instruction for adding another element of the array, so that the statement actually uses the address of the next element each time. Since the elements of the array are stored one after the other in memory, it means that, as mentioned above, we must increase the address of the element being added by 1 in each loop run. This task is carried out by statements placed on lines 8, 9 and 10. The accumulator receives from the memory the instruction of adding another element of the array (line 6 in the program), incremented by 1 and re-written in the same place to the memory. Thanks to this, when this instruction (from line 6) is executed again, its argument will be another element of the array. It should be clearly stated that the modification from lines 8 - 10 only changes the content of the instructions from line 6 and does not change the meaning of the `Array` label. If in the program in several instructions it would be necessary to refer to the next elements of the array in the subsequent runs of the loop, all these instructions should be modified in a similar way.

The task of adding elements to an array can also be done a bit differently.

```
1. loop:   POB n
2.         ODE One
3.         SOM end
4.         ŁAD n
5.         POB Pob0
6.         DOD Addr
7.         ŁAD Instr
8.         POB Sum
9. Instr:  RPA   // here is the instruction to add another
                 // element of the array
10.        ŁAD Sum
11.        POB Addr
12.        DOD One
13.        ŁAD Addr
14.        SOB loop
15. end:   POB Sum
16.        STP
17. n:     RST 3
18. Array: RST 10
19.        RST -5
20.        RST 3
21. Sum:   RST 0
22. One:   RST 1
23. Pob0:  POB 0
24. Addr:  RST Array
```

And in this case, further elements of the array are added in the loop, but the code modification is slightly different. This solution resembles a solution using pointers in C. The location marked with the label `Addr` (line 24) contains the address of the beginning of the array (it is like an pointer of this array). In each loop run, the value of the `Array` element indicated by the `Addr` pointer (line 9) is added to the current `Sum`. The current value of the `Addr` location is added to the `POB 0` instruction code (line 23) and the result is saved in place marked with the `Instr` (line 9) label. In this way, we obtain the instruction adding to the accumulator the content of the appropriate memory location, the address of which contains the `Addr` variable.

I/O Operations

Previous programs have not exchanged information with the environment but only processed the data initially stored in memory and saved it there (in memory). These programs would be much more general and would be more useful if the input could be entered from the keyboard and displayed on the screen. To make this possible, however, it is necessary to complete the list of instructions on the so-called input/output instructions.

Input/output instructions

To ensure the exchange of information with the environment, at least two instructions must be added: instruction for reading a single character from the input device (e.g. keyboard) and instruction enabling the output of the accumulator content to the output device (e.g. screen).

The arguments of both of these instructions will be the device number to which the instruction applies. We assume that the standard input device (keyboard) is marked with the number 1, and the standard output device is marked with the number 2. The instruction for entering the character will be called WPR. After entering, the code of the character is placed in the accumulator. The character output instruction (WYP) will print on the output device with the given number the character whose code is in the accumulator. Using these instructions, you can, for example, write a program that copies characters entered on the keyboard to the output device. The program in the loop reads the character and writes it to the standard output. The signal to terminate the program may be the reading of an appointment ending. With us such a character will be a space. Below is the code for this program.

```
loop:   WPR 1
        ODE Space
        SOZ end
        DOD Space
        WYP 2
        SOB loop
end:    STP
Space:  RST 32
```

Typically, computing programs do not need individual characters as input data, but numbers saved as a sequence of digits. Also numerical results should be displayed as decimal digits. This means that it is necessary to convert the loaded strings of digits to the number and reverse conversion of the number written in binary to the appropriate sequence of decimal digits. It seems that it will be best if these tasks are implemented in the form of special subroutines.

Subroutines

Using subroutines means you need to add more instructions to your processor. The subroutine call (SDP) instruction, similar to a simple jump, causes the transition to execute the instruction whose address is an argument to the SDP instruction. In addition, the content of the program counter (so-called trace) is stored on the stack. By retaining the trace on the stack, it is possible to return to the next instruction after the SDP instruction at the moment when the subroutine was completed. This return is carried out by the PWR instruction. In addition, we will introduce two more instructions related to the stack: DNS - placing the accumulator value on the top of the stack and PZS removing the value from the top of the stack and storing it in the accumulator. These instructions are often used to pass parameters between the calling program and the subroutine. Below is a subroutine that calculates the square of a given number. The number whose square we want to find should first be placed in the accumulator. Also in the accumulator the subroutine will return the result.

```
1.         SDP Square      // calculate the square of the number
                           // stored in the accumulator
2.         STP             // after executing the subprogram result in
                           // the accumulator, we finish the program
3. Square: ŁAD Param       // here the subroutine begins, the
                           // parameter stored in the variable Param
4.         SOM Negat       // when negative, we calculate the square
                           // with its absolute value
5.         SOZ zero        // when zero, you do not have to count
                           // anything (the result is zero), return
6. Posit:  ŁAD Product     // non-negative and not zero, so positive;
                           // first run of the loop
7.         ŁAD Param       // needed when there was a negative number
                           // and its absolute value was calculated
8. loop:   ODE One         // in every loop we decrement the counter
9.         SOZ GoBack      // when the counter reaches zero, we finish
10.        ŁAD Counter     // when it's not over, you need to remember
                           // the counter
11.        POB Product     // in every loop run, to the product we add
12.        DOD Param       // a number whose square we count
13.        ŁAD Product
14.        POB Counter     // and then again decrement the counter
15.        SOB loop        // and check whether it is not over
16. GoBack:POB Product     // the result should be in the accumulator
17. zero:  PWR             //
18. Negat: ODE Param       // parameter with a negative number, so we
                           // first calculate its absolute value
19.        ODE Param       // and then we do the same as with
20.        SOB Posit       // the positive number (-a)^2 = a^2
21. Param: RPA             // value of the subroutine parameter
22. Product: RPA           // result of calculating the square
23. Counter: RPA           // loop pass counter
24. One:   RST 1
```

The main program (lines 1 and 2) has been simplified here to a maximum and is limited to calling a subroutine that calculates the square of the number stored in the accumulator. The content of the subroutine starts on line 3. The parameter (the number whose square you want to count) is in the accumulator. First, the subprogram checks what parameter value it has to deal with. If it is zero, immediately go to the return instruction from the subroutine and the returned result is 0. When the accumulator has a negative number, the opposite number is calculated (or if someone prefers the absolute value) and then proceed in the same way as for the positive number. In the case of a positive number, to determine its square, the number should be added a certain number of times. This task implements a loop, the content of which was written in lines 8 to 15.

Subroutine of writing an integer on the output device

The task of writing a number placed in the accumulator on the output device can be done in the form of the Write subprogram with the code presented below.

```
             SDP Write
             STP
Write:       ŁAD Number
             POB Zero
             DNS
             POB Number
             SOM Abs
Posit:       DZI Ten
             MNO Ten
             ŁAD Tmp
             POB Number
             ODE Tmp
             DOD Char0
             DNS
             POB Tmp
             DZI Ten
             SOZ End
             ŁAD Number
             SOB Posit
Abs:         POB Minus
             WYP 2
             POB Zero
             ODE Number
             ŁAD Number
             SOB Posit
End:         PZS
             SOZ GoBack
             WYP 2
             SOB End
GoBack:      PWR
Zero:        RST 0
Number:      RPA
Tmp:         RPA
Ten:         RST 10
Char0:       RST '0'
Minus:       RST '-'
```

The list of instructions has been extended by instructions for multiplication (MNO) and division (DZI). The basic task of this procedure is to convert a binary number to a sequence of decimal digits (represented as ASCII characters). This conversion was carried out using the stack. On the stack, numbers are loaded one by one (stored with the use of ASCII code) being the next residues from division of the binary number by 10. To each residue the ASCII code corresponding to the character 0 is added, thanks to which the conversion to the characters of this code is made immediately. Numbers forming the form of numbers are therefore generated from the end. Thanks to the use of the stack, however, they will be output to the inverted, and thus correct, order. In the next step, the number is divided by 10 and the actions are repeated until the value reaches 0. Then from the stack we take the numbers one by one and print them out to the output device. To easily detect when the digits end, before they were written to the stack there was a so-called sentinel (here in the form of the number 0, see lines 4 and 5 in the subprogram code). Picking a sentinel from the stack results in the ending of outputting digits to the output and thus marks the end of the subroutine.

Subprogram of entering numbers

Also when entering a number, the next characters (digits) are converted to the appropriate binary form. This task is performed by the Read procedure. It reads subsequent digits forming the form of a natural number and converts to a binary character until it reaches a sign other than a digit. After completion, the loaded number is located in the accumulator. We leave the number entering code to the reader for independent analysis.

```
1.                SDP Read
2.                STP
3. Read:          POB Zero
4. GoBack:        ŁAD Number
5.                WPR 1
6.                ODE Char0
7.                SOM Ready
8.                ODE Ten
9.                SOM Skip
10. Ready:        POB Number
11.               PWR
12. Skip:         DOD Ten
13.               ŁAD Digit
14.               POB Number
15.               MNO Ten
16.               DOD Digit
17.               SOB GoBack
18. Digit:        RPA
19. Number:       RPA
20. Ten:          RST 10
21. Zero:         RST 0
22. Char0:        RST '0'
```

Lab tasks

Simple programs

1. Write a program calculating the greatest common divisor of two natural numbers placed in memory locations labeled A and B

2. Write a program calculating the lowest common multiple of two natural numbers placed in memory locations labeled A and B
3. Write a program calculating the product of two natural numbers placed in memory locations labeled A and B
4. Write a program calculating the rest of the division of two natural numbers placed in memory locations labeled A and B
5. There is an n-element array labeled Arr. Write a program to check how many times the value specified in the Pattern memory location appears in this array.
6. There is an n-element array labeled Arr. Write a program that changes the elements of this array in pairs. Assume that n is an even number. For example, if in the array there were consecutively the numbers 1 2 3 and 4, then after the program is to be 2 1 4 3.
7. There is an n-element array labeled Arr. Write a program that reverses the contents of this array. For example, if the array was successively numbers 1 2 3 4 and 5, then after the program is to be 5 4 3 2 1.
8. There is an n-element array labeled Arr. Write a program that finds the value of the largest element of this array and inserts it into the memory location marked with the Max label.
9. There is an n-element array labeled Arr. Write a program that finds the value of the smallest element in this array and inserts it into the memory location labeled Min.

Input/output

1. Write a program reading two natural numbers and writing out the smaller one.
2. Write a program that loads two natural numbers and prints out the larger one.
3. Write a program that reads two natural numbers and prints their greatest common denominator.
4. Write a program that reads two natural numbers and prints their lowest common multiple.
5. Write a program that reads two natural numbers and prints their product.
6. Write a program that reads two natural numbers and prints the rest of the division of these two numbers.
7. Write a program that reads a string from a standard input and writes only the numbers from this string to the standard output
8. Write a program that reads a string from a standard input and writes the string to the standard output but with small letters converted into capitals
9. Write a program printing out the encrypted text read from the input. Use a one-dimensional affine code

Subroutines

1. Write a program calculating the factorial of a natural number using the subroutine executing the multiplication of integers.
2. Write a program that calculates the power of two natural numbers using the subroutine executing the multiplication of integers.
3. Write a program calculating the value of the determinant computed from the elements of square matrix 2x2 using the subprogram executing the multiplication of integers.

4. Write a program to check if a given number is a prime number using the sieve of Eratosthenes method using a subprogram to find the remainder of the division of integers