Development of the Game Hangman in Assembly Programming Language

Stefan Tešanović and Predrag Mitrović

***Abstract* — This paper describes the realization of the Hangman game in Microsoft Visual Studio, using the assembler programming language, Kip Irvine's and MASM libraries. Knowledge of the work in the software tool is demonstrated, as well as advanced knowledge of assembler and work with library functions. The obtained game uses less CPU time than its realizations in other higher level programming languages.**

***Keywords* — irvine32, masm, assembly, hangman, game**

I. Introduction

THIS paper provides an algorithm for the realization of popular computer game Hangman [1]. The basic idea of this work was the development of Hangman game using Microsoft Visual Studio 2015 programming environment and libraries used for easier implementation of the game in assembly programming language.

Assembler (assembly) language is a low-level programming language, and it is specific to a particular processor (microcontroller) architecture [2]. As the oldest programming language, and of all languages, it bears the closest resemblance to a native machine language. It provides a direct access to computer hardware, requiring you to understand much about your computer's architecture and operating system [3].

A large number of modern microcontrollers are based on a similar architecture as the microcontroller 8086 [3] and have similar abbreviations for the names of assembly instructions. So, the study of microcontroller 8086 is reckoned not only for pedagogical reasons but also for usability [4].

Through numerous examples of research being studied through lectures, students are led to a correct conclusion that when designing microcontrollers it is necessary to know the algorithm which the hardware will use and that it is necessary to know the hardware on which the software will be executed to write an optimal algorithm [2] – [8]. So, both parts of the design should be projected in parallel.

Today's software for digital signal processing is based on some high-level programming language, but optimization of code execution is achieved writing individual functions in assembly [9]. We can say that the assembly is used when we need fast and efficient processing of the signal.

The Irvine 32 library [3] makes programming in assembly easier and provides libraries for work with strings and support for MS-Windows features. Some of the main supporting features are graphical environment functions.

In this paper, we will describe designing the Hangman game using the x86 architecture with Irvine 32 libraries [3] in the programming environment Microsoft Visual Studio 2015.

The paper is organized as follows. In section II the x86 processor architecture is described in detail. Section III provides the game specification. In section IV are described functions and concepts used for developing the game. Concluding remarks are given in section V.

II. Processor x86 architecture

Unlike programming in higher programming languages, where it is necessary to understand the endpoints of the procedural or object-oriented programming, for programming in assembler it is necessary to understand hardware.

Assembly language is not portable, because it is designed for a specific processor family. Thus, there are many different assembly languages widely used today, each based on a processor family [3].

Of all the hardware, the most important thing to us is the architecture of the central processor unit (CPU). In the CPU, where all calculations and logical operations take place, there are a limited number of storage locations named registers, a high-frequency clock, a control unit and an arithmetic logic unit [2] – [3].

In this paper, we have decided to use x86 processor for development of the project. In order to understand our realization of the game, it is necessary to get acquainted with the processor architecture, especially with registers and the way the strings are processed.

At our disposal, we have general-purpose registers. The general-purpose registers are primarily used for arithmetic and data movement. As shown in Fig. 1, the lower 16 bits of the EAX register can be referenced by the name AX. Portions of some registers can be addressed as 8-bit values. For example, the AX register has an 8-bit upper half named AH and an 8-bit lower half named AL [3].

Multiplication and division instructions automatically use EAX register. It is often called the extended accumulator register [2] – [3].

Paper received May 29, 2018; revised November 25, 2018; accepted November 26, 2018. Date of publication December 25, 2018. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Nataša Nešković.

This paper is a revised and expanded version of the paper presented at the 25th Telecommunications Forum TELFOR 2017 [10].

Stefan Tešanović and Predrag Mitrović with School of Electrical Engineering, University of Belgrade, Bul. kralja Aleksandra 73, 11120 Belgrade, Serbia (e-mail: stefan@stefan.engineer, predrag.mitrovic@yahoo.com).

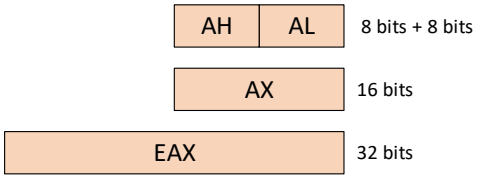

Fig. 1. General-purpose registers.

The same overlapping relationship that exists for the EAX registers also exists for EBX, ECX, and EDX registers.

High-speed memory transfer instructions use ESI and EDI registers. They are sometimes called the extended source index and extended destination index registers [2] – [3].

These six registers represent the core of understanding how the instructions are executed in the assembler and the communication between the memory and the processor.

As we do the processing of strings in the project, it is necessary to emphasize that in the register ECX is the number of iterations of the corresponding instruction for working with strings.

III. Description of the Game Realization

The game is realized using the already known term which is to be guessed. For our purpose, we chose that the number of terms should be at least 10. At the beginning of the game, the player tries to guess the term by writing letters from the standard input. If the entered letter does not exist in the specified term, then the part of Hangman is added. However, if the letter exists in the term which a player tries to guess, then the letter is written to the appropriate position in the word. If the letter has already been hit, the user is informed necessarily about it, and if the user by accident writes it, then it does not affect the game. Every time a new letter is entered from the standard input, it is necessary to print it on the screen. When a user guesses the current term, they are allowed to hit the new one. If the user misses the letters enough times, the entire Hangman is drawn out, and a player loses the game. The game can be interrupted by pressing the ESC button.

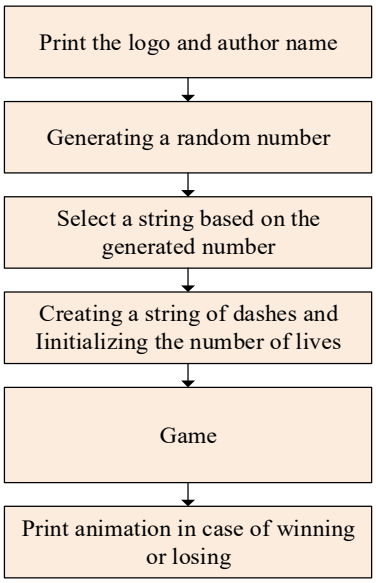

Fig. 2. A simplified program diagram.

The description of the project code and the principle of the game operation are described through the program flow diagram depicted in Fig. 2. In diagram from Fig. 2, each of the segments will be explained in separate subsections.

A. Print the logo and name of the author

At the very beginning of the program, we print the logo as a series of strings. A variable *endl* represents a string in which we typed two characters that represent the transition to a new line. In the variable *messageSize*, the length of this string is saved.

To print the string, we use the *WriteConsole* function [8]. It prints a series of characters on the screen starting from the current position of the cursor and moves the cursor to the position of the last printed character. In order for this function to be used before its use, it is necessary to take access to the standard input or output. With the *GetStdHandle* function [8], we take a standard input and output handle, which can then use some of the following functions: *ReadFile*, *WriteFile*, *ReadConsoleInput*, *WriteConsole* or *GetConsoleScreenBufferInfo* [8]. All mentioned functions can be found in Irvine libraries [3], [6] – [8].

B. Generating a random number

Generating a random number we have achieved the power of two Irvine functions [3], [6-8]. The *Randomize* function restarts the randomly generated number based on the current time. If we did not use this function, we would always have the same random number. The *RandomRange* function generates an unsigned pseudo-random 32-bit integer in the range 0 to *N*-1. In our case, it generates a number from 0 to 9. The code that realizes this is shown in Fig. 3.

```
;Part of code for generate random number from 0 until 9
  mov  eax,10          ;get random 0 to 9
  call Randomize       ;re-seed generator
  call RandomRange
  mov  ranNum,eax      ;save random number
```

Fig. 3. Code generating a random number.

C. Choosing the word which we guess

Based on the randomly generated number, we need to select one of the ten words, which we have previously defined. Fig. 4 shows the declaration of the string of all the words that can be selected.

```
;All words what is posible to guess.
;Pick by random generartor and put in selectedWords
manyWords   BYTE "BICYCLE", 0
            BYTE "CANOE", 0
            BYTE "SCATEBOARD", 0
            BYTE "OFFSIDE", 0
            BYTE "TENNIS", 0
            BYTE "SOFTBALL", 0
            BYTE "KNOCKOUT", 0
            BYTE "CHALLENGE", 0
            BYTE "SLALOM", 0
            BYTE "MARATHON", 0
            BYTE 0                  ; End of list
len equ $ - manyWords
```

Fig. 4. Define words what to guess.

It should be noted that by adding zeros at the end of each word we have made a null-terminated string. This is important for using Irvine Library. All the Irvine functions which work with a string can do only with the null-terminated string, meaning that the zero at the end of string marks the end of this word.

In the main program, it is necessary to place an accidentally generated number in the EDX registry and call the *find_str* function. Code for the *find_str* function is displayed in Fig 5. The function returns the EDI, the pointer to the string. After we get a pointer to the given word, we need to copy that word into the variable *selectWords* that represent the selected word that we are looking. We used Irvine's *Str_Copy* library for copy string [3], [6] – [8].

```
find_str PROC                ; ARG: EDX = index
    lea edi, manyWords       ; Address of string list

    mov ecx, len             ; Maximal number of bytes to scan
    xor al, al               ; Scan for 0

    @@:
    sub edx, 1
    jc done                  ; No index left to scan = string found
    repne scasb              ; Scan for AL
    jmp @B                   ; Next string

  done:
    ret

find_str ENDP                ; RESULT: EDI pointer to string[edx]
```

Fig. 5. Code of find_str function.

Fig. 6 shows the code in the main program for copying the word to which the EDI registry displays in the variable *selectedWords*. In order to be able to select the selected word in this way, it was necessary to go to the declaration after each word. The function from the Irvine [3] library *Str_copy* copies the string from the current position to which the EDI register reaches 0, i.e., the *Str_copy* function copies the null-terminated string from the source location to the new location [6] – [8].

```
;Copy find world in variable selectedWords
  INVOKE Str_copy,
      ADDR [edi],
      ADDR selectedWords
```

Fig. 6. Apply *Str_copy* function.

D. Creating a string of dashes

After we have chosen the word we are guessing, we need to form a series of dashes that will be of the same length as the chosen word. On the screen, we will print all the dashes in the first interaction, and later as the player will enter the letters, replace the corresponding dashes with the letters in the appropriate places as well as in the selected word.

In the main program, it is necessary to call the *make_array_dash* function that creates a series of dashes, code of this realization is given in Fig. 7.

At the beginning of the function, it is necessary to determine the length of the word that we have chosen. We did this by using the Irvine [3] function *StrLength* which returns the length of null-terminated stored in the EDX register [6-8]. After raising it, the length of the word is stored in the EAX register. After that, we use the instruction set to store the corresponding number of dashes in the variable *guessWords*. For the further work with strings, it is necessary to add 0 to the end of the string *guessWords*, in

```
make_array_dash PROC
    mov  edx,OFFSET selectedWords
    call StrLength           ; Length of a null-terminated string pointed to by
    mov  lengthArray,eax

    mov al, '-'              ; Default charcter for guessWords
    mov ecx, lengthArray     ; REP counter
    mov edi, offset guessWords  ; Destination
    rep stosb                ; Build guessWords
    mov BYTE PTR [edi], 0    ; Store the null termination

    ret
make_array_dash ENDP
```

Fig. 7. Code of *make_array_dash* function.

order to make it null-terminated and to make easier the search and work with strings.

The replacement of certain dashes with letters is done within the part of the game and will be described in a separate chapter.

E. Initializing the number of lives and drawing figures

One of the requirements when designing this project is to print the Hangman on the screen (standard command output), i.e., figures. Therefore, we decided to keep the number of player's lives in variable *statusGameLive* and to depend on the number of lives draw the appropriate figure. Drawing the figure was done by using *print_hangman_live*. The *print_hangman_live* function is extremely simple. It simply checks the value of *statusGameLive* and depending on its value jumps to a specific label.

After we have determined on which label we jump, we make a drawing of the Hangman, write the word that we find in the form of dashes (the letters that are affected are shown) and a series of letters that user has entered so far separated by a comma. The print that we realize is depicted in Fig. 8.

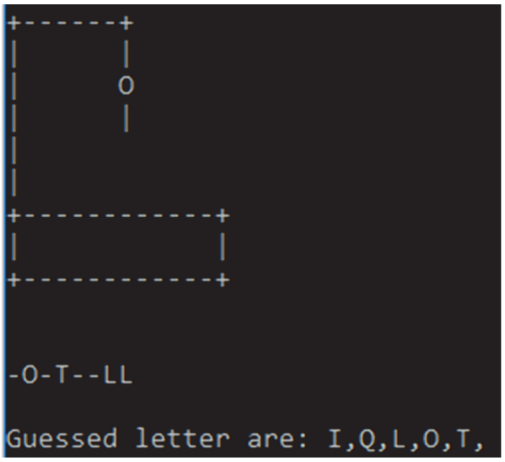

Fig. 8. Display of prints on standard output.

IV. Realization of the Game

The diagram that represents the core of this project and combines several functions in it is presented in Fig. 9. The beginning of the game is based on the printing of the Hangman to standard input. After that, it is necessary to check how much lives are remaining to the player. In the case when there are no lives left, it is necessary to jump on the *loop_game_over*, we start the animation for the loss of the game for one minute, and then we leave the game. Otherwise, we will continue the game. The variable *statusGameLive* stores the number of lives.

Fig. 10 shows the code used to load letters from the keyboard. The first thing to check after loading letters from

the standard input is whether it is ESC. In case the ESC key is pressed, i.e., the entered character has a value of 27, it is necessary to suspend the execution of the game. Otherwise, we continue to execute the program.

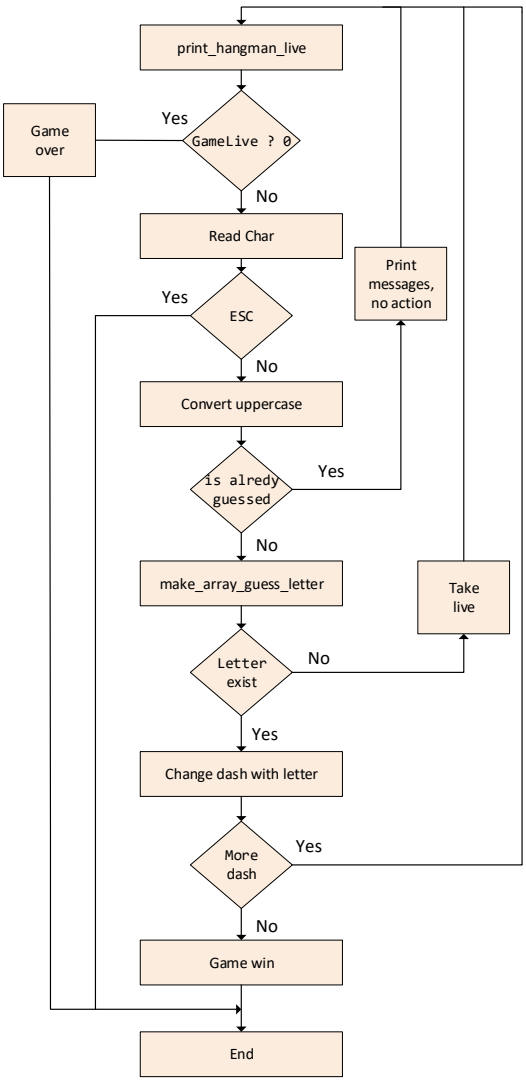

Fig. 9. Game flow diagram.

```
    mov  eax,green+(black*16)
    call SetTextColor

    mWrite <"Guess a letter: ">

    call readChar   ;User inputs char
    cmp al, 27      ;Check if is press ESC
    je exit_main    ;YES, end game
    and al, 0DFH    ;Convert lowercase input to uppercase.
                    ;If uppercase, it remains uppercase
    push eax
    sub al, 'A'     ;checks if it is a letter
    cmp al, 'Z'-'A'
    jbe uppercase
    jmp again_input_world
uppercase:
    pop eax
    mov guessLetter, al
    call WriteChar
    call Crlf       ;new line
    call Crlf       ;new line

    mov  eax,white+(black*16)
    call SetTextColor
```

Fig. 10. Code for the letter loading realization.

Since the variables (terms) are consisting exclusively of capital letters, it is necessary to examine whether a large or a small letter is entered. If a small letter is entered, the program converts it to a capital letter. Also in this part of the program, a message is printed to enter the letter and the printout of the entered letter itself. The print is green in order to be more visible to the player.

Next, it is necessary to check whether the entered letter has already been hit or not. In the variable *guessLetterArray*, all letters entered from the standard input are stored. In case the letter has already been hit, it is necessary to jump *loop_guess_letter_exists* over the loop, print the corresponding message and return to the beginning to re-enter the letters. The code that prints the message to the standard output is shown in Fig. 11.

```
loop_guess_letter_exists:

        mov  eax,red+(black*16)
        call SetTextColor

        mWrite <"Sorry, you alredy guessed letter, ">
        mov al, guessLetter
        call WriteChar
        call Crlf                       ; new line
        mWrite <"I repeat you one more time the letter what you guessed. ">
        call Crlf                       ; new line
        mWrite <"Guessed letter are: ">
        mov edx, offset guessLetterArray
        call WriteString                ; write a string pointed to by EDX
        call Crlf                       ; new line
        call Crlf                       ; new line

        mov  eax,white+(black*16)
        call SetTextColor

        jmp again_input_world           ; Guess next letter
```

Fig. 11. Code for printing message that letter has already been hit.

In case the letter has not been hit yet, it is necessary to add it to the string *guessLetterArray*. This is accomplished by calling the *make_array_guess_letter* function. The function is shown in Fig. 12.

```
make_array_guess_letter PROC
    mov  edx, OFFSET guessLetterArray
    call StrLength              ; Length of a null-terminated string pointed to by EDX
    mov  lengthArray, eax

    mov edi, offset guessLetterArray ; Destination
    add edi, lengthArray
    mov al, guessLetter
    mov BYTE PTR [edi], al      ; Store guessLetter
    inc edi
    mov BYTE PTR [edi], ','     ; Store the null termination

    ret
make_array_guess_letter ENDP
```

Fig. 12. Code to add letters entered into a string of entered characters.

It is also necessary to check if the entered letter is in the selected word, the selection of which is described in the chapter above. The search string is similar to a bit earlier and is shown in Fig. 13.

```
;Check if letter is in selectedWords. If not take life
mov ecx, LENGTHOF selectedWords
mov edi, offset selectedWords
mov al, guessLetter                 ; Load character to find
repne scasb                         ; Search
jne loop_take_live                  ; Letter exist take life
```

Fig. 13. Check whether the selected word contains the entered letter.

If the selected word does not contain the entered letter, we scroll *loop_take_live* over the loop, reducing the value of the variable *statusGameLive* to one, i.e., we take one life to the player and return to the beginning of that game.

If the guess word contains an entered letter, then it is necessary to place the letter in the appropriate place in the *guessWords* strand, i.e., to replace the corresponding dash with letters in the same places as they are in *selectedWords*. The code that performs this realization is shown in Fig. 14.

```
; We are making new array, guess letter whange dash on right pleace
  mov esi, offset selectedWords       ; Source
  mov edi, offset guessWords          ; Destination
  mov ecx, LENGTHOF selectedWords     ; Number of bytes to check
  mov al, guessLetter                 ; Search for that character
  xor ebx, ebx                        ; Index EBX = 0

ride_hard_loop:
    cmp [esi+ebx], al                 ; Compare memory/register
    jne @F                            ; Skip next line if no match
    mov [edi+ebx], al                 ; Hang 'em lower
    @@:
    inc ebx                           ; Increment pointer
    dec ecx                           ; Decrement counter
    jne ride_hard_loop                ; Jump if ECX != 0
```

Fig. 14. Replace dashes with the letter inserted.

In the next step, we need to check if there are some other dashes in the *guessWords* strand or we hit all the letters. If *guessWords* contain dashboards, we return to the beginning of the game. Otherwise, we print the animation, and we end the game. The code for this realization is shown in Fig. 15.

V. Conclusion

In this paper, we describe a realization of the game Hangman. Although the game is well known and built numerous times, our realization gives an exciting approach to a programming assignment made entirely in assembly programming language. This realization is especially interesting for educational purposes because it provides an exciting way to introduce students to assembly programming and architecture of x86 processors.

```
;Is there more letter to guess of we finish
  mov ecx, LENGTHOF guessWords
  mov edi, offset guessWords
  mov al, letterDash                  ; Load character to fin
  repne scasb                         ; Search
  jne loop_game_win                   ; No more letter
  jmp again_input_world               ; Guess next world
```

Fig. 15. Check if there are still unchecked letters.

Appendix

All codes of this realization can be downloaded from GitHub by following link https://goo.gl/cfaWH6.

Acknowledgment

We devote this work to professors, mentors dr Aleksandra Lekić and dr Milan Prokin, family and Giovanni who was always there for us with advice and support.

References

[1] https://en.wikipedia.org/wiki/Hangman_(game)
[2] Carter, Paul A. *PC Assembly Language*. Lulu.com, 2007.
[3] Irvine, Kip R. *Assembly language for Intel-based computers*. Prentice Hall, 2003.
[4] Prokin, M. *Računarska elektronika*. Akademska misao, 2005
[5] Hyde, R. *The art of assembly language*. No Starch Press, 2010.
[6] http://kipirvine.com/asm/index6th.htm
[7] http://www.asmirvine.com/
[8] http://programming.msjc.edu/asm/help/index.html?page=source%2Fabout.htm
[9] Kuo, Sen M., Bob H. Lee, and Wenshun Tian. *Real-time digital signal processing: fundamentals, implementations and applications*. John Wiley & Sons, 2013.
[10] S. Tešanović and P. Mitrović, *Development of the Game Hangman in Assembly Programming Language*, 2017 25th Telecommunications Forum (TELFOR), Belgrade, 2017, pp. 912-915.