A PROJECT REPORT ON GUESSING GAME USING ASSEMBLY LANUGUAGE IN TASM (8086 MICROPROCESSOR)

BAHROZ JAVED

ID#CSC-18F-134

SECTION: B

SEMESTER: 4TH

UNDER THE GUIDENCE: MRS. SEHER JAMANI

CONTENT OF PROJECT REPORT

1) INTRODUCTION OF ASSEMBLY LANGUAGE
2) BLOCK DIAGRAM
3) TYPES OF ASSEMBLY LANGUAGE
4) GUESSING GAME
5) FLOWCHART OF PROJECT
6) SOURCE CODE
7) OUTPUT SCREENSHOT
8) CONCLUSION OF PROJECT
9) REFERANCE

INTRODUCTION ABOUT ASSEMBLY LANGUAGE

An assembly language is an extremely low-level programming language that has a 1-to-1 correspondence to machine code, the series of binary instructions which move values in and out of registers in a CPU (or another microprocessor). A microprocessor is a mechanical calculator. It has a number of named registers, which are like holding pens for numbers. It receives instructions in the form of machine code, which is represented by a series of binary bits (1s and 0s).

Each assembly language is specific to a particular computer architecture and sometimes to an operating system. However, some assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. In contrast to assembly languages, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling. Assembly language may also be called *symbolic machine code.*

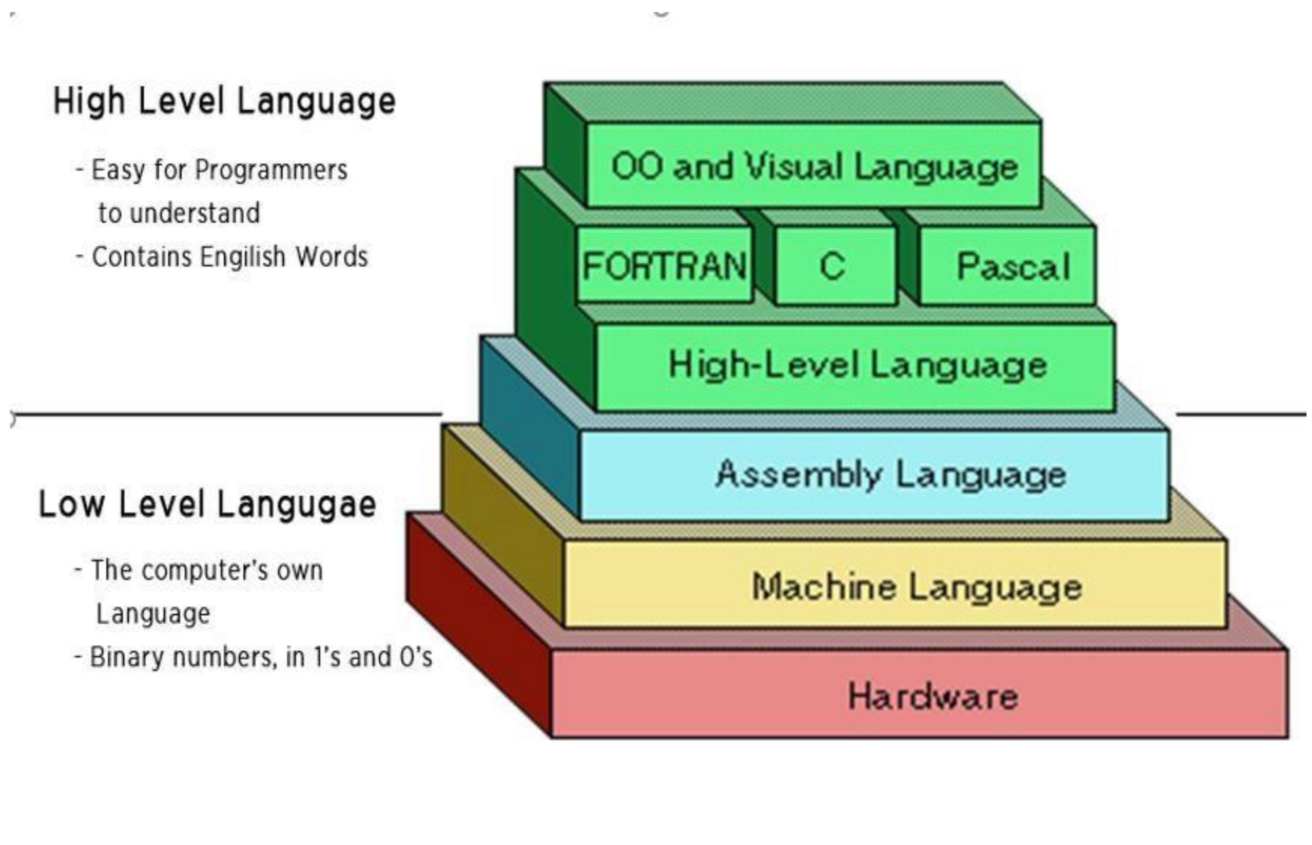
The assembler language is the symbolic programming language that lies closest to the machine language in form and content. The assembler language is useful when You need to control your program closely, down to the byte and even the bit level.

A programming language that is once removed from a computer's machine language. Machine languages consist entirely of numbers and are almost impossible for humans to read and write. Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers.
Each type of CPU has its own machine language and assembly language, so an assembly language program written for one type of CPU won't run on another. In the early days of programming, all programs were written in assembly language. Now, most programs are written in a high-level language such as FORTRAN or C. Programmers still use assembly language when speed is essential or when they need to perform an operation that isn't possible in a highlevel language.

Assembly language uses a mnemonic to represent each low-level machine instruction or opcode, typically also each architectural register, flag, etc. Many operations require one or more operands in order to form a complete instruction. Most assemblers permit named constants, registers, and labels for program and memory locations, and can calculate expressions for operands. Thus, the programmers are freed from tedious repetitive calculations and assembler programs are much more readable than machine code. Depending on the architecture, these elements may also be combined for specific instructions or addressing modes using offsets or other data as well as fixed addresses. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an *operation code* ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution – e.g., to generate common short sequences of instructions as inline, instead of *called* subroutines.

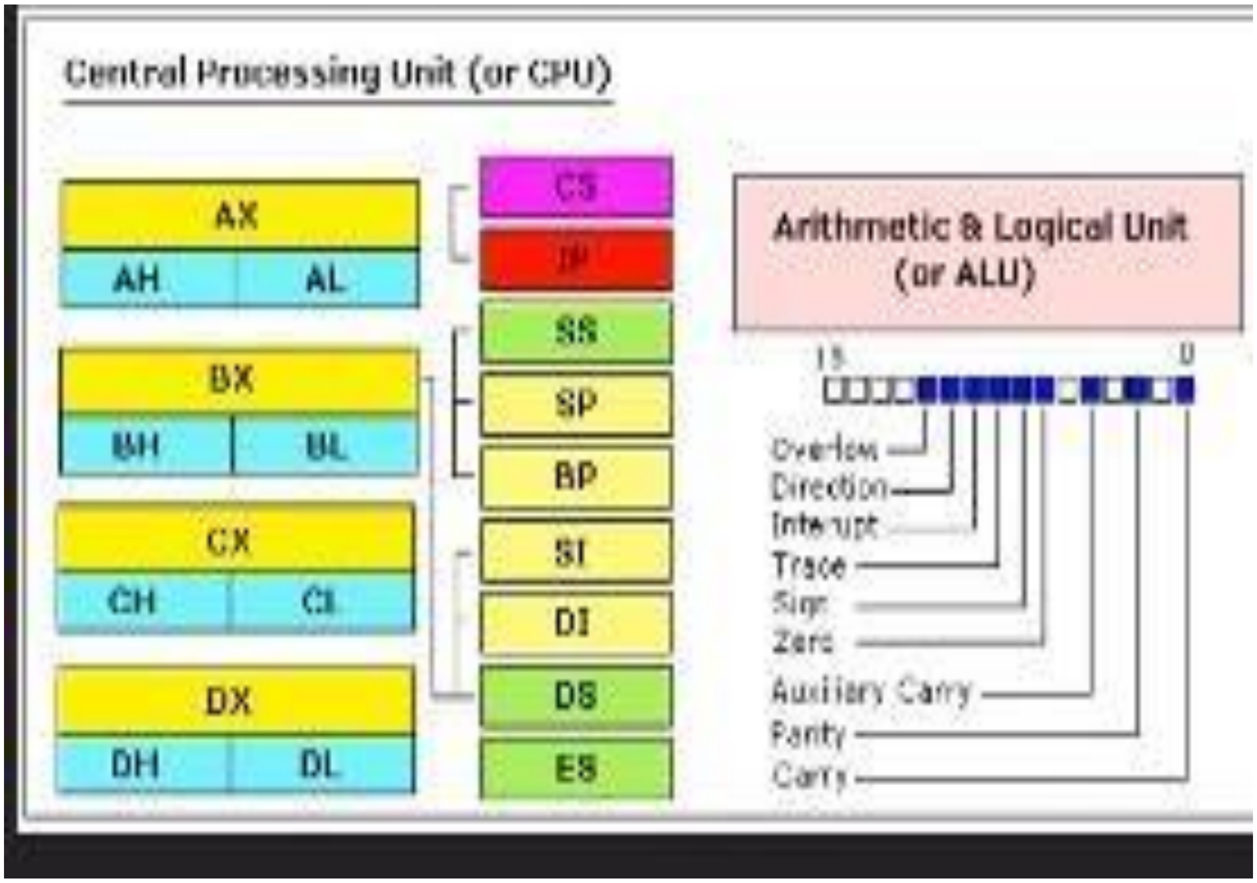
Some assemblers may also be able to perform some simple types of instruction setspecific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearrangement or insertion of instructions, such as some assemblers for RISC architectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.



Today, assembly language is used primarily for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are device drivers, low-level embedded systems, and real-time systems.

Assembly language is as close to the processor as you can get as a programmer so a welldesigned algorithm is blazing -- assembly is great for speed optimization. It's all about performance and efficiency. Assembly language gives you complete control over the system's resources. Much like an assembly line, you write code to push single values into registers, deal with memory addresses directly to retrieve values or pointers.

Memory address resister’s



General Purpose Registers (a.k.a. scratch registers)

- AX (AH,AL) Accumulator : Main arithmetic register
- BX (BH,BL) Base : Generally used as a memory base or offset
- CX (CH,CL) Counter : Generally used as a counter for loops
- DX (DH,DL) Data : General 16-bit storage, division remainder

Offset Registers

- IP Instruction pointer : Current instruction offset
- SP Stack pointer : Current stack offset
- BP Base pointer : Base for referencing values stored on stack
- SI Source index : General addressing, source offset in string ops

- DI Destination index : General addressing, destination in string ops

Segment Registers

- CS Code segment : Segment to which IP refers
- SS Stack segment : Segment to which SP refers
- DS Data segment : General addressing, usually for program's data area
- ES Extra segment : General addressing, destination segment in string ops

Flags Register (Respectively bits 11,10,9,8,7,6,4,2,0)

- OF Overflow flag : Indicates a signed arithmetic overflow occurred
- DF Direction flag : Controls incr. direction in string ops (0=inc, 1=dec)
- IF Interrupt flag : Controls whether interrupts are enabled
- TF Trap flag : Controls debug interrupt generation after instructions
- SF Sign flag : Indicates a negative result or comparison
- ZF Zero flag : Indicates a zero result or an equal comparison
- AF Auxiliary flag : Indicates adjustment is needed after BCD arithmetic
- PF Parity flag : Indicates an even number of 1 bits
- CF Carry flag : Indicates an arithmetic carry occurred4

The general purpose registers AX, BX, CX, and DX are 16-bit registers but each can also be used as two separate 8-bit registers. For example, the high (or upper) byte of AX is called AH and the low byte is called AL. The same H and L notation applies to the BX, CX, and DX. Most instructions allow these 8-bit registers as operands.

Registers AX, BX, CX, DX, SI, DI, BP, and SP can be used as operands for most instructions. However, only AX, BX, CX, and DX should be used for general purposes since SI, DI, BP, and SP are usually used for addressing.

Types of Assembly Languages

- **CISC: Complex Instruction-Set Computer** Developed when people wrote assembly language Complicated, often specialized instructions with many effects Examples from x86 architecture String move, Procedure enter, leave Many, complicated addressing modes So complicated, often executed by a little program (microcode) Examples: Intel x86, 68000, PDP-11

- **RISC: Reduced Instruction-Set Computer**Response to growing use of compilers Easier-to-target, uniform instruction sets “Make the most common operations as fast as possible” Load-store architecture: Arithmetic only performed on registers, Memory load/store instructions for memory-register transfers Designed to be pipelined Examples: SPARC, MIPS, HP-PA, PowerPC

- **<u>DSP: Digital Signal Processor</u>** designed specifically for signal processing algorithms Lots of regular arithmetic on vectors Often written by hand Irregular architectures to save power, area Substantial instruction-level parallelism Examples: TI 320, Motorola 56000, Analog Devices

- **<u>VLIW Assembly Language</u>** Response to growing desire for instruction-level parallelism Using more transistors cheaper than running them faster Many parallel ALUs Objective: keep them all busy all the time Heavily pipelined More regular instruction set Very difficult to program by hand Looks like parallel RISC instructions Examples: Itanium, TI 320C60000

GUESSING GAME

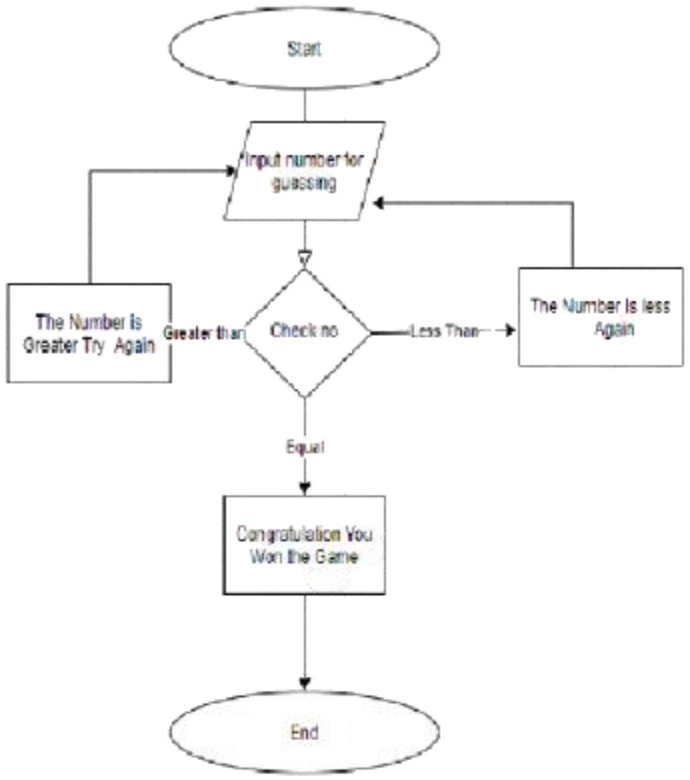
The computer is going to randomly select an integer from 1 to 15. You'll keep guessing numbers until you find the computer's number, and the computer will tell you each time if your guess was too high or too low: Once you've found the number, reflect on what technique you used when deciding what number to guess next.

Maybe you guessed 1, then 2, then 3, then 4, and so on, until you guessed the right number. We call this approach **linear search**, because you guess all the numbers as if they were lined up in a row. It would work. But what is the highest number of guesses you could need? If the computer selects 15, you would need 15 guesses. Then again, you could be really lucky, which would be when the computer selects 1 and you get the number on your first guess. How about on average? If the computer is equally likely to select any number from 1 to 15, then on average you'll need 8 guesses. But you could do something more efficient than just guessing 1, 2, 3, 4, …, right? Since the computer tells you whether a guess is too low, too high, or correct, you can start off by guessing 8. If the number that the computer selected is less than 8, then because you know that 8 is too high, you can eliminate all the numbers from 8 to 15 from further consideration. If the number selected by the computer is greater than 8, then you can eliminate 1 through 8. Either way, you can eliminate half the numbers. On your next guess, eliminate half of the remaining numbers. Keep going, always eliminating half of the remaining numbers.

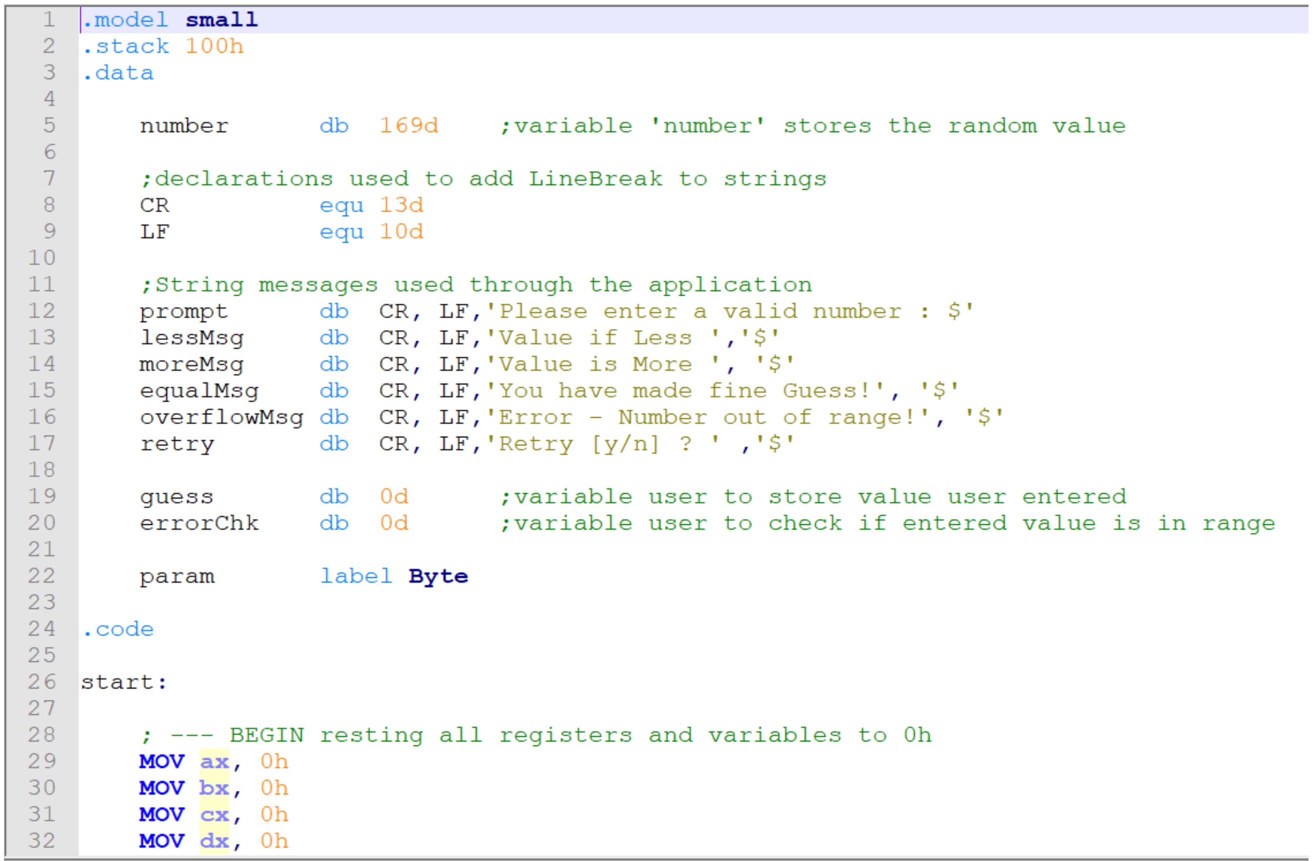
We call this halving approach **binary search**, and no matter which number from 1 to 15 the computer has selected, you should be able to find the number in at most 4 guesses with this technique.

Here, try it for a number from 1 to 300. You should need no more than 9 guesses.

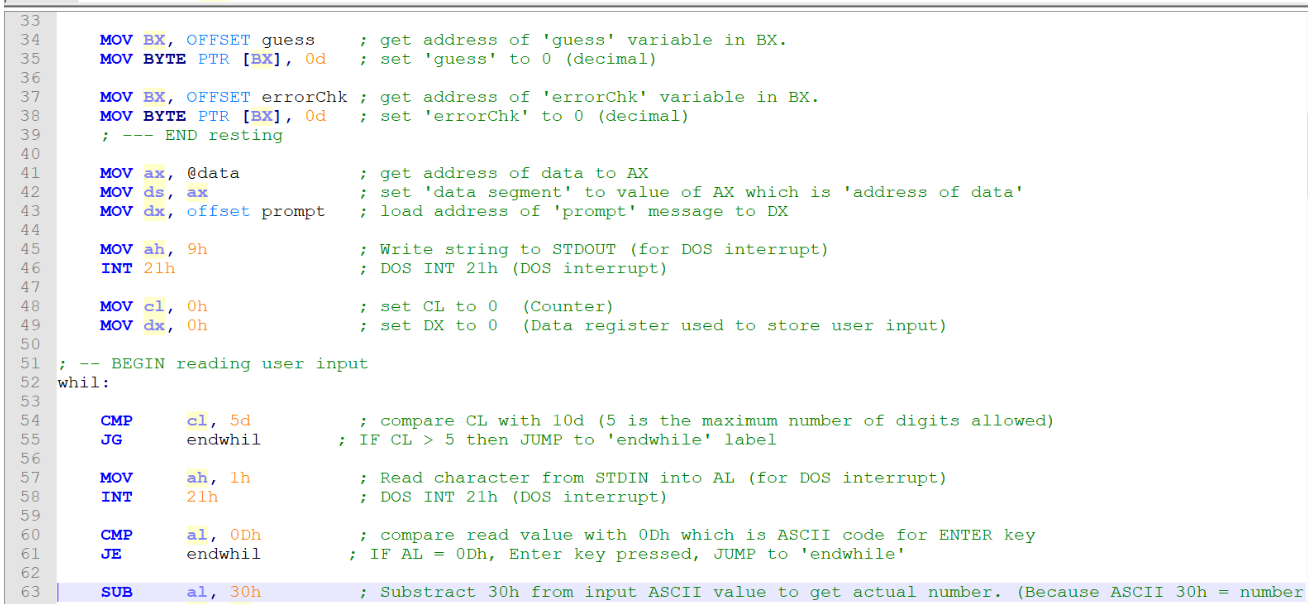
FLOWCHART


Start
Input number for guassing
The Number Is Greater Try Again
Greater than
Check no
Less Than
The Number Is less Again
Equal
Congratulation You Won the Game
End

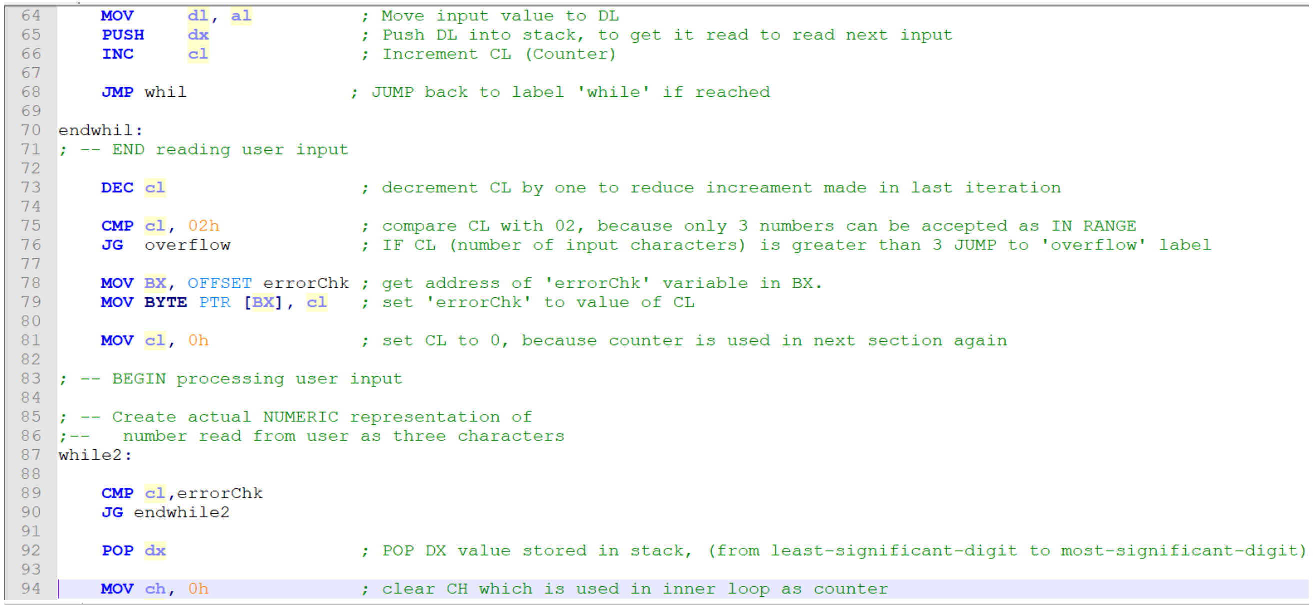
SOURCE CODE.



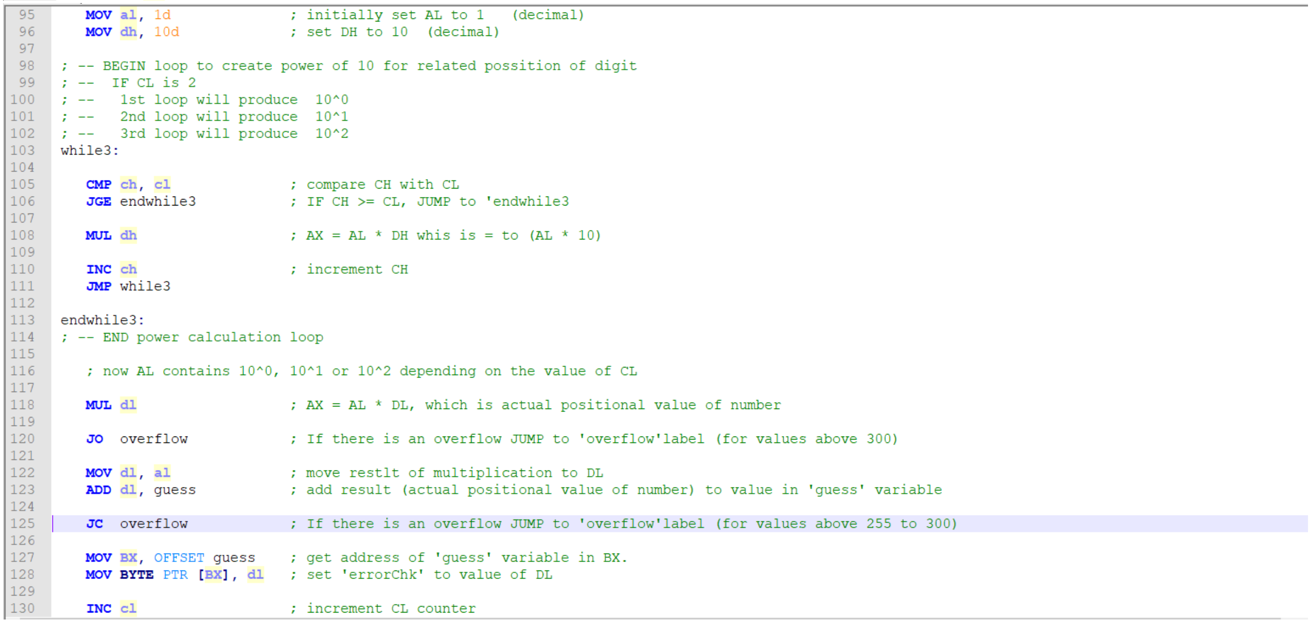
```
.model small
.stack 100h
.data

    number       db  169d    ;variable 'number' stores the random value

    ;declarations used to add LineBreak to strings
    CR           equ 13d
    LF           equ 10d

    ;String messages used through the application
    prompt       db  CR, LF,'Please enter a valid number : $'
    lessMsg      db  CR, LF,'Value if Less ','$'
    moreMsg      db  CR, LF,'Value is More ', '$'
    equalMsg     db  CR, LF,'You have made fine Guess!', '$'
    overflowMsg db  CR, LF,'Error - Number out of range!', '$'
    retry        db  CR, LF,'Retry [y/n] ? ' ,'$'

    guess        db  0d      ;variable user to store value user entered
    errorChk     db  0d      ;variable user to check if entered value is in range

    param        label Byte

.code

start:

    ; --- BEGIN resting all registers and variables to 0h
    MOV ax, 0h
    MOV bx, 0h
    MOV cx, 0h
    MOV dx, 0h
```



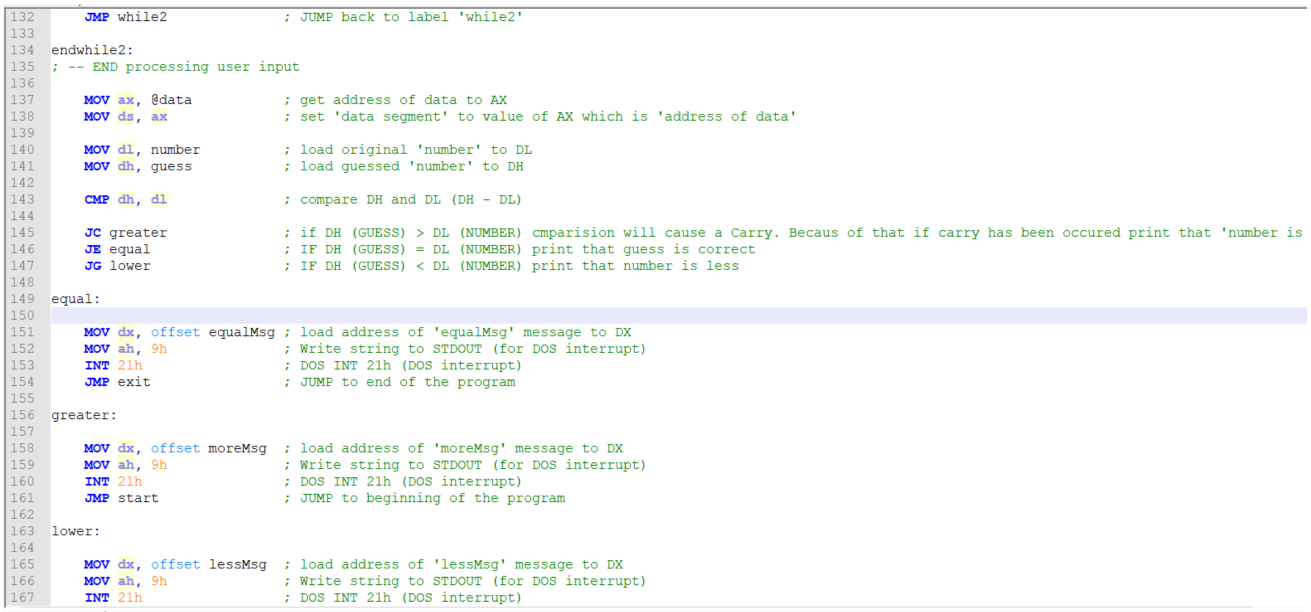
```

    MOV BX, OFFSET guess    ; get address of 'guess' variable in BX.
    MOV BYTE PTR [BX], 0d   ; set 'guess' to 0 (decimal)

    MOV BX, OFFSET errorChk ; get address of 'errorChk' variable in BX.
    MOV BYTE PTR [BX], 0d   ; set 'errorChk' to 0 (decimal)
    ; --- END resting

    MOV ax, @data           ; get address of data to AX
    MOV ds, ax              ; set 'data segment' to value of AX which is 'address of data'
    MOV dx, offset prompt   ; load address of 'prompt' message to DX

    MOV ah, 9h              ; Write string to STDOUT (for DOS interrupt)
    INT 21h                 ; DOS INT 21h (DOS interrupt)

    MOV cl, 0h              ; set CL to 0  (Counter)
    MOV dx, 0h              ; set DX to 0  (Data register used to store user input)

; -- BEGIN reading user input
whil:

    CMP     cl, 5d          ; compare CL with 10d (5 is the maximum number of digits allowed)
    JG      endwhil       ; IF CL > 5 then JUMP to 'endwhile' label

    MOV     ah, 1h          ; Read character from STDIN into AL (for DOS interrupt)
    INT     21h             ; DOS INT 21h (DOS interrupt)

    CMP     al, 0Dh         ; compare read value with 0Dh which is ASCII code for ENTER key
    JE      endwhil        ; IF AL = 0Dh, Enter key pressed, JUMP to 'endwhile'

    SUB     al, 30h         ; Substract 30h from input ASCII value to get actual number. (Because ASCII 30h = number
```



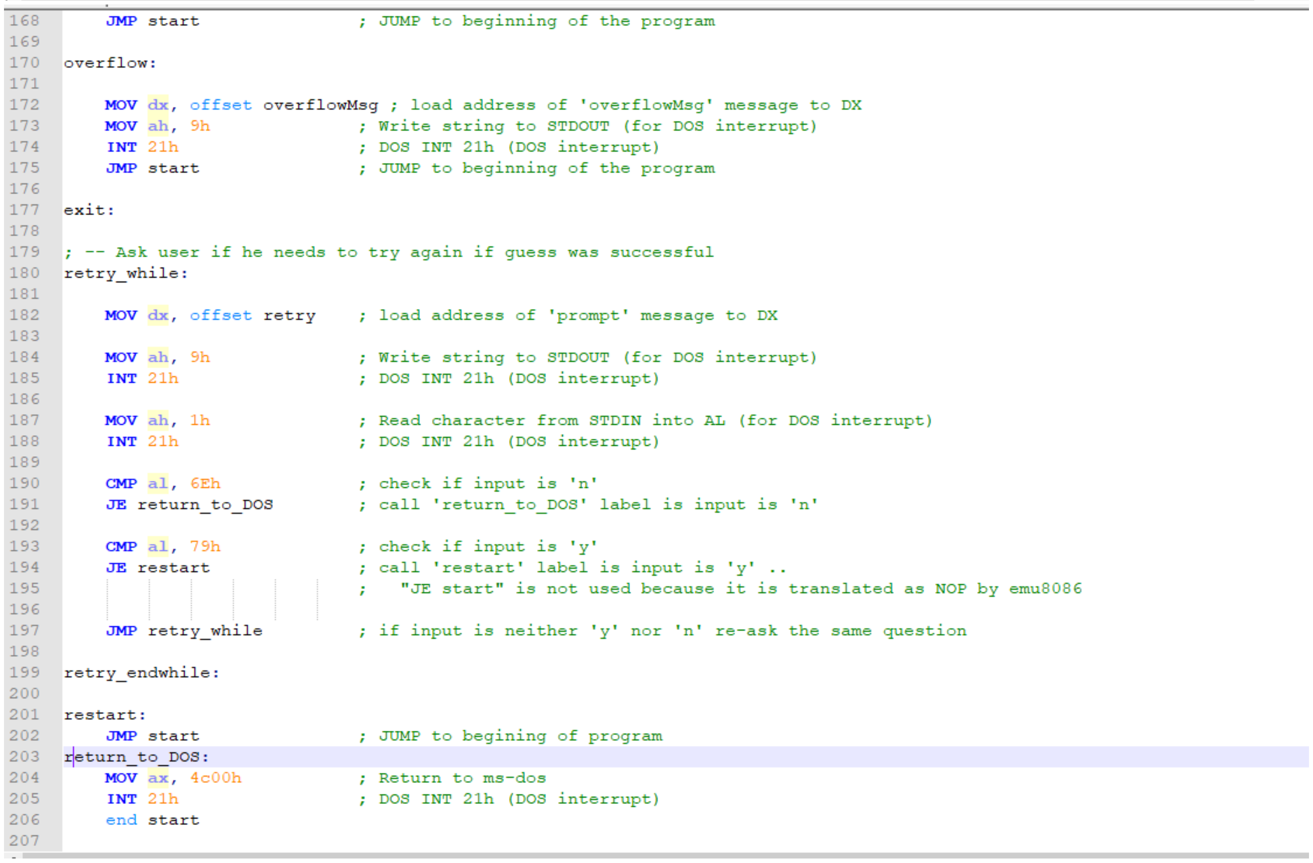
```
    MOV     dl, al          ; Move input value to DL
    PUSH    dx              ; Push DL into stack, to get it read to read next input
    INC     cl              ; Increment CL (Counter)

    JMP whil               ; JUMP back to label 'while' if reached

endwhil:
; -- END reading user input

    DEC cl                  ; decrement CL by one to reduce increament made in last iteration

    CMP cl, 02h             ; compare CL with 02, because only 3 numbers can be accepted as IN RANGE
    JG  overflow            ; IF CL (number of input characters) is greater than 3 JUMP to 'overflow' label

    MOV BX, OFFSET errorChk ; get address of 'errorChk' variable in BX.
    MOV BYTE PTR [BX], cl   ; set 'errorChk' to value of CL

    MOV cl, 0h              ; set CL to 0, because counter is used in next section again

; -- BEGIN processing user input

; -- Create actual NUMERIC representation of
;--   number read from user as three characters
while2:

    CMP cl,errorChk
    JG endwhile2

    POP dx                  ; POP DX value stored in stack, (from least-significant-digit to most-significant-digit)

    MOV ch, 0h              ; clear CH which is used in inner loop as counter
```



```
    MOV al, 1d              ; initially set AL to 1   (decimal)
    MOV dh, 10d             ; set DH to 10  (decimal)

; -- BEGIN loop to create power of 10 for related possition of digit
; --   IF CL is 2
; --    1st loop will produce  10^0
; --    2nd loop will produce  10^1
; --    3rd loop will produce  10^2
while3:

    CMP ch, cl              ; compare CH with CL
    JGE endwhile3           ; IF CH >= CL, JUMP to 'endwhile3

    MUL dh                  ; AX = AL * DH whis is = to (AL * 10)

    INC ch                  ; increment CH
    JMP while3

endwhile3:
; -- END power calculation loop

    ; now AL contains 10^0, 10^1 or 10^2 depending on the value of CL

    MUL dl                  ; AX = AL * DL, which is actual positional value of number

    JO  overflow            ; If there is an overflow JUMP to 'overflow'label (for values above 300)

    MOV dl, al              ; move restlt of multiplication to DL
    ADD dl, guess           ; add result (actual positional value of number) to value in 'guess' variable

    JC  overflow            ; If there is an overflow JUMP to 'overflow'label (for values above 255 to 300)

    MOV BX, OFFSET guess    ; get address of 'guess' variable in BX.
    MOV BYTE PTR [BX], dl   ; set 'errorChk' to value of DL

    INC cl                  ; increment CL counter
```

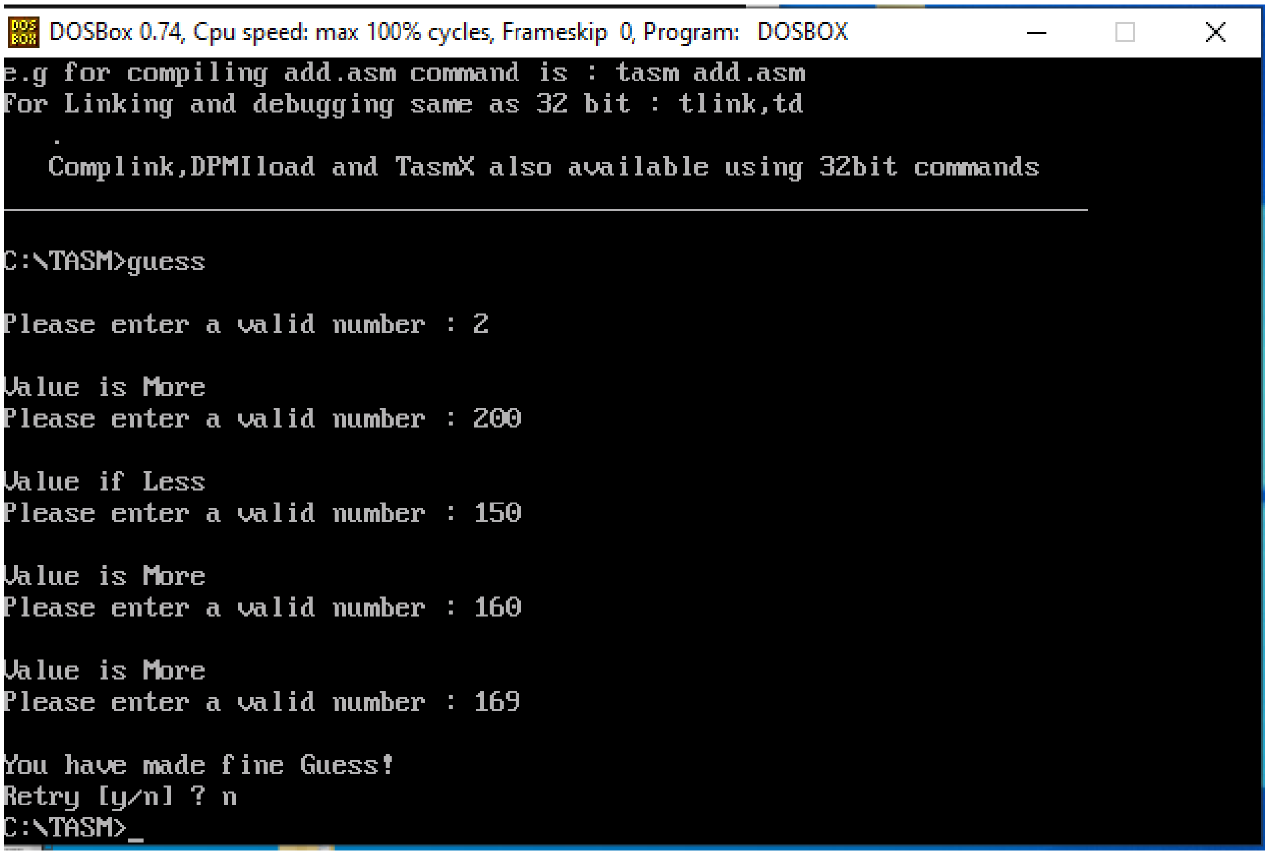


```
    JMP while2              ; JUMP back to label 'while2'

endwhile2:
; -- END processing user input

    MOV ax, @data           ; get address of data to AX
    MOV ds, ax              ; set 'data segment' to value of AX which is 'address of data'

    MOV dl, number          ; load original 'number' to DL
    MOV dh, guess           ; load guessed 'number' to DH

    CMP dh, dl              ; compare DH and DL (DH - DL)

    JC greater              ; if DH (GUESS) > DL (NUMBER) cmparision will cause a Carry. Becaus of that if carry has been occured print that 'number is
    JE equal                ; IF DH (GUESS) = DL (NUMBER) print that guess is correct
    JG lower                ; IF DH (GUESS) < DL (NUMBER) print that number is less

equal:

    MOV dx, offset equalMsg ; load address of 'equalMsg' message to DX
    MOV ah, 9h              ; Write string to STDOUT (for DOS interrupt)
    INT 21h                 ; DOS INT 21h (DOS interrupt)
    JMP exit                ; JUMP to end of the program

greater:

    MOV dx, offset moreMsg  ; load address of 'moreMsg' message to DX
    MOV ah, 9h              ; Write string to STDOUT (for DOS interrupt)
    INT 21h                 ; DOS INT 21h (DOS interrupt)
    JMP start               ; JUMP to beginning of the program

lower:

    MOV dx, offset lessMsg  ; load address of 'lessMsg' message to DX
    MOV ah, 9h              ; Write string to STDOUT (for DOS interrupt)
    INT 21h                 ; DOS INT 21h (DOS interrupt)
```



```
    JMP start                  ; JUMP to beginning of the program

overflow:

    MOV dx, offset overflowMsg ; load address of 'overflowMsg' message to DX
    MOV ah, 9h                 ; Write string to STDOUT (for DOS interrupt)
    INT 21h                    ; DOS INT 21h (DOS interrupt)
    JMP start                  ; JUMP to beginning of the program

exit:

; -- Ask user if he needs to try again if guess was successful
retry_while:

    MOV dx, offset retry       ; load address of 'prompt' message to DX

    MOV ah, 9h                 ; Write string to STDOUT (for DOS interrupt)
    INT 21h                    ; DOS INT 21h (DOS interrupt)

    MOV ah, 1h                 ; Read character from STDIN into AL (for DOS interrupt)
    INT 21h                    ; DOS INT 21h (DOS interrupt)

    CMP al, 6Eh                ; check if input is 'n'
    JE return_to_DOS           ; call 'return_to_DOS' label is input is 'n'

    CMP al, 79h                ; check if input is 'y'
    JE restart                 ; call 'restart' label is input is 'y' ..
                               ;    "JE start" is not used because it is translated as NOP by emu8086

    JMP retry_while            ; if input is neither 'y' nor 'n' re-ask the same question

retry_endwhile:

restart:
    JMP start                  ; JUMP to begining of program
return_to_DOS:
    MOV ax, 4c00h              ; Return to ms-dos
    INT 21h                    ; DOS INT 21h (DOS interrupt)
    end start

```

OUTPUT SCREENSHOT: -


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: DOSBOX

```
e.g for compiling add.asm command is : tasm add.asm
For Linking and debugging same as 32 bit : tlink,td
   .
   Complink,DPMIload and TasmX also available using 32bit commands
______________________________________________________________________

C:\TASM>guess

Please enter a valid number : 2

Value is More
Please enter a valid number : 200

Value if Less
Please enter a valid number : 150

Value is More
Please enter a valid number : 160

Value is More
Please enter a valid number : 169

You have made fine Guess!
Retry [y/n] ? n
C:\TASM>_
```

CONCLUSION

In this Project we have successfully build a simple number guessing game in assembly language using microprocessor 8086

REFERENCES

1) Assembly Language Programming Tutorial, a very thorough 55video series on assembly, following the book *Assembly Language for x86 Processors (6th Edition)* by Kip Irvine (if you aren't following the videos, you'll probably want the more recent edition)
2) Assembly Language Programming Video Course, a 70-part video series, taught by Arthur Griffith, who has a very folksy charm
3) X86 instruction listings, full list of all instructions for the x86 architectures, with notes on when each was added
4) X86 Opcode and Instruction Reference
5) Intel X86 Assembly Language Cheat Sheet (pdf)
6) IEEE-CALVIS32: assembly language visualizer and simulator for intel x86-32 architecture BY Jennica Grace Alcalde.
7) WIKIPEDIA ASSEMBLY LANGUAGE x86.
8) IBM-knowledge center on assembly language.