A PROJECT REPORT ON CALCULATOR AND BODY MASS INDEX USING ASSEMBLY LANGUAGE IN EMULATOR(8086)

HARSH CHOKSHI

ID#92479

UNDER THE GUIDENCE: PROF. Dr. QING ZHU

CONTENT OF PROJECT REPORT

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.

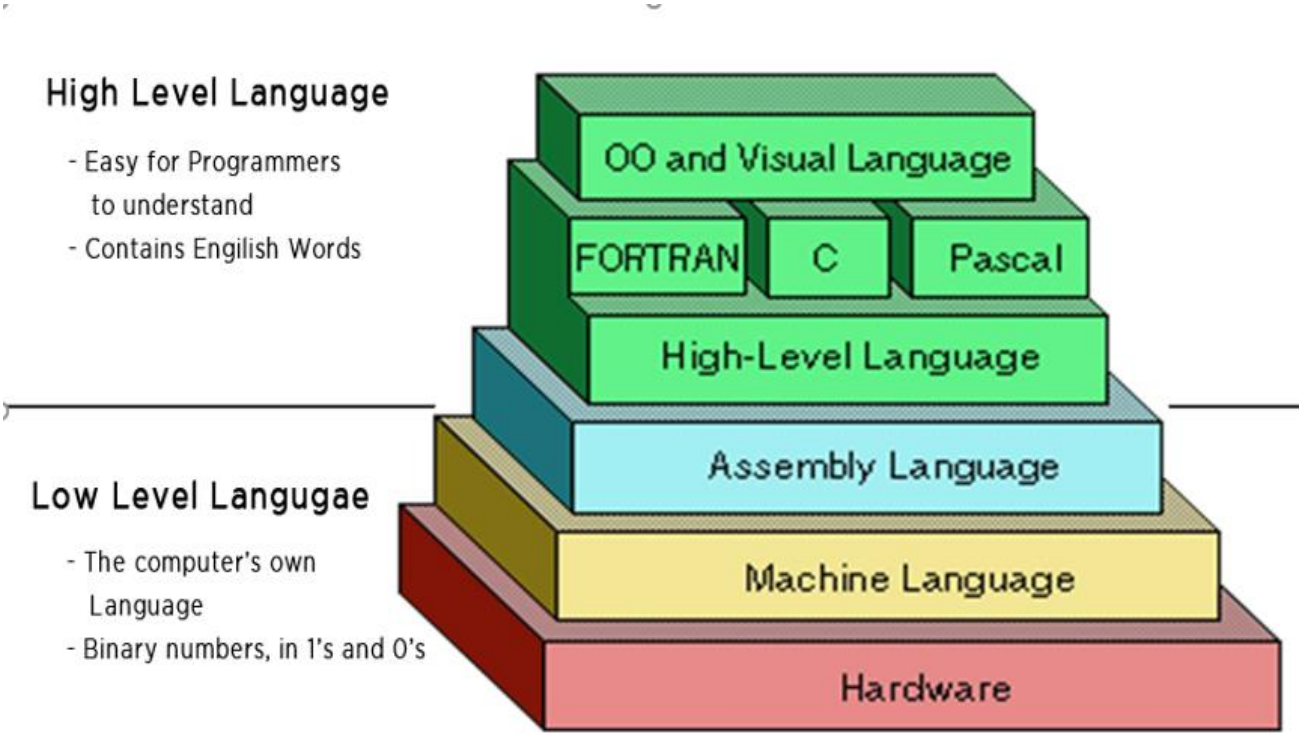
A programming language that is once removed from a computer's machine language. Machine languages consist entirely of numbers and are almost impossible for humans to read and write. Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use 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 high-level 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.

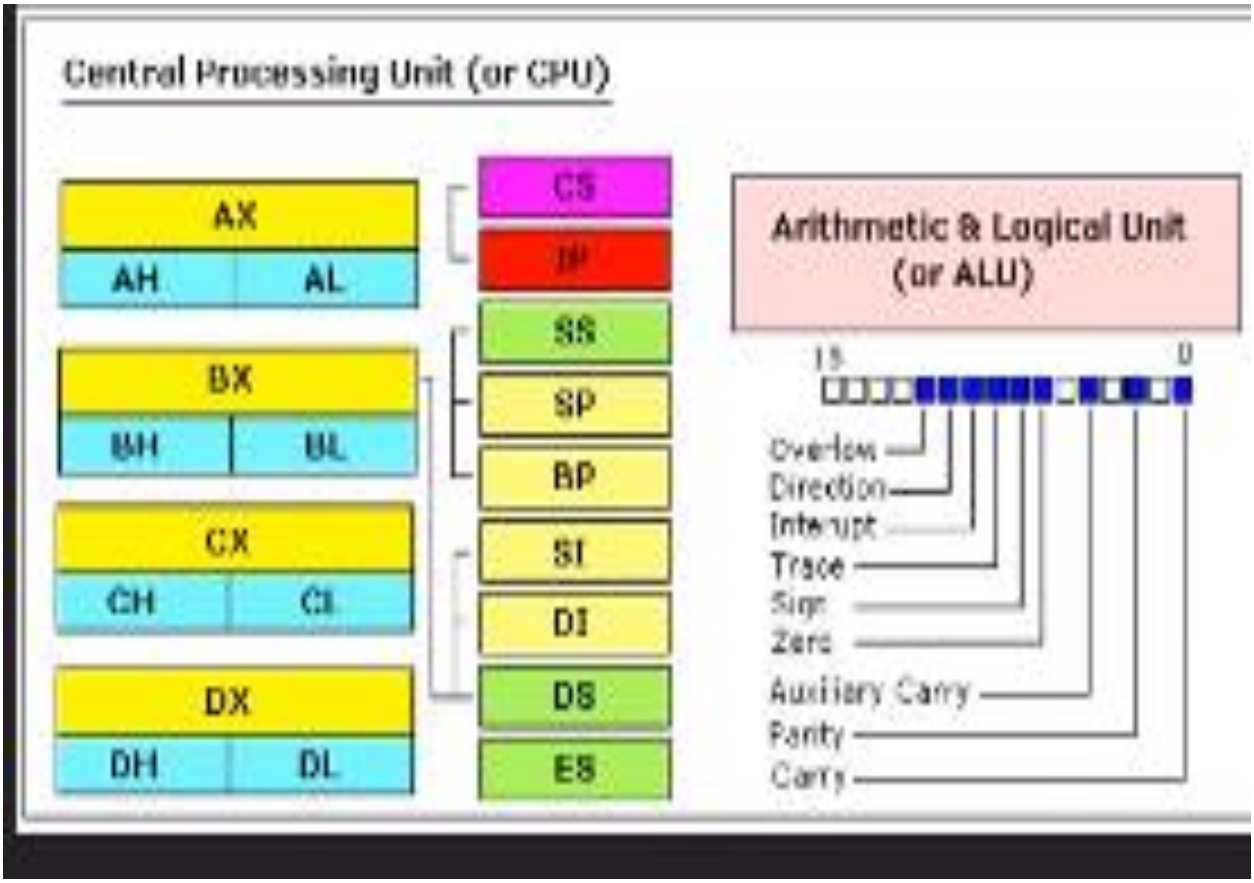
Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) 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 well-designed 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
- **DSP: Digital Signal Processor** 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

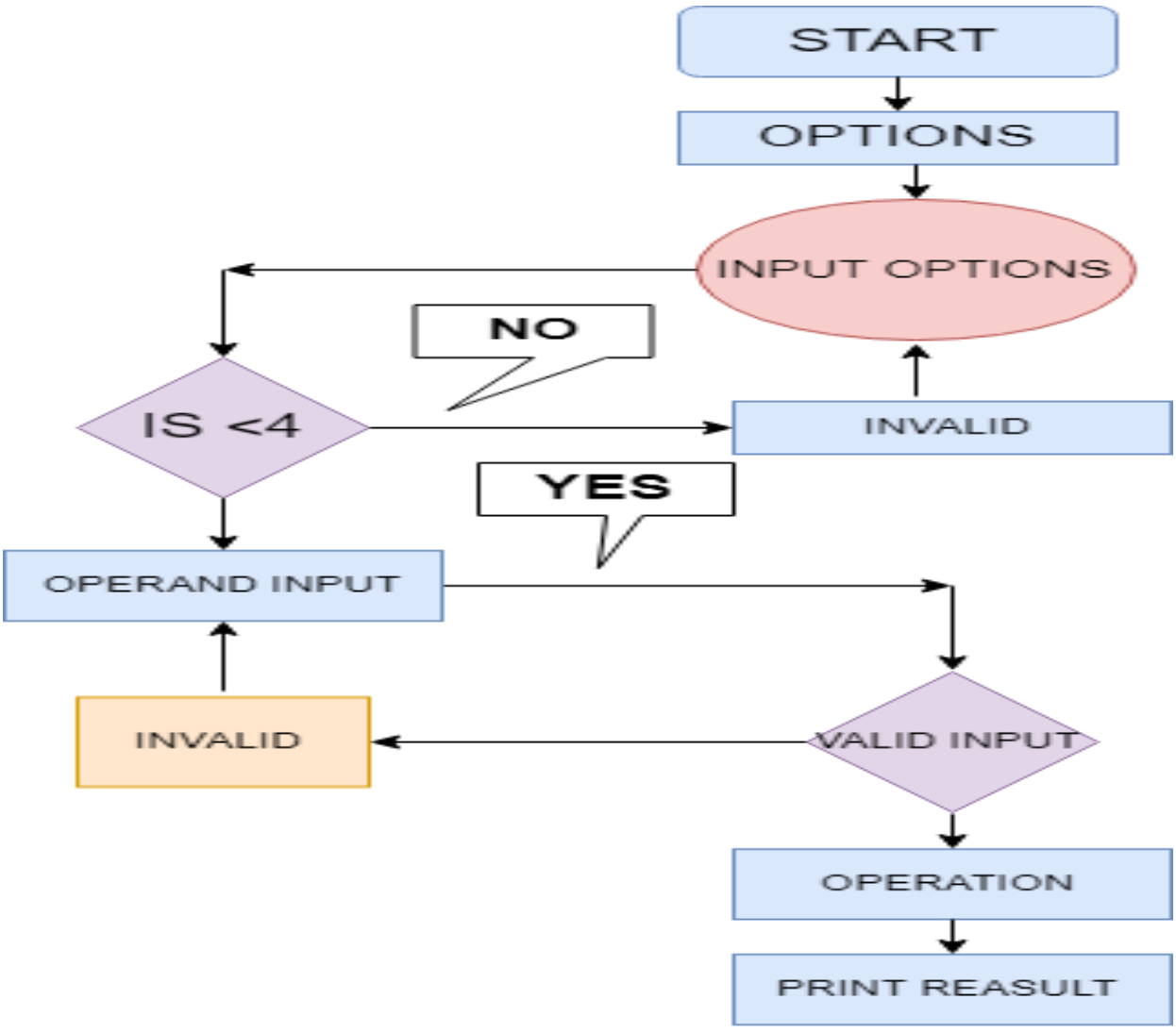
- **VLIW Assembly Language** Response to growing desire for instruction-level parallelism Using more transistors cheaper than running them faster Many parallel ALUs Objective: keep them all busy all the time Heavily pipelined More regular instruction set Very difficult to program by hand Looks like parallel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

CALCULATOR

An **electronic calculator** is typically a portable electronic device used to perform calculations, ranging from basic arithmetic to complex mathematics. A calculator is a device that performs arithmetic operations on numbers. calculators can do only addition, subtraction, multiplication, and division which is pretty basic.

in this project we have computed the assembly language code(8086) to do the basic function's, I have simulated the programmed in emulator software. This is a microprocessor emulator with an integrated 8086 Assembler. The emulator can run programs on a Virtual Machine, and emulate real hardware including screen, memory, and input and output devices. It helps you program in assembly language. The source code is compiled by assembler and then executed on Emulator step-by-step, allowing you to watch registers, flags and memory while your program runs.

FLOWCHART



EMULATOR SOFTWARE

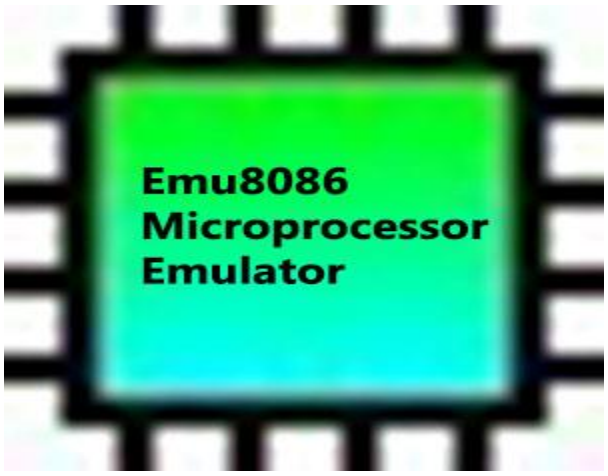
8086 Microprocessor Emulator, also known as EMU8086, is an emulator of the program 8086 microprocessor. It is developed with a built-in 8086 assembler. This application is able to run programs on both PC desktops and laptops. This tool is primarily designed to copy or emulate hardware. These include the memory of a program, CPU, RAM, input and output devices, and even the display screen.

There are instructions to follow when using this emulator. It can be executed into one of the two ways: backward or forward. There are also examples of assembly source code included. With this, it allows the programming of assembly language, reverse engineering, hardware architecture, and creating miniature operating system (OS).

The user interface of 8086 Microprocessor Emulator is simple and easy to manage. There are five major buttons with icons and titles included. These are “Load”, “Reload”, “Step Back”, “Single Step”, and “Run”. Above those buttons is the menu that includes “File”, “View”, “Virtual Devices”, “Virtual Drive”, and “Help”. Below the buttons is a series of choices that are usually in numbers and codes. At the leftmost part is an area called “Registers” with an indication of either “H” or “L”. The other side is divided into two, which enables users to manually reset, debug, flag, etc.

Visual Microprocessor Emulator - All in one tool to study Assembly Language: Integrated Assembler, CPU Emulator

- Interactive Debugger and Disassembler.
- Advanced Source Editor with Syntax Highlight.
- Complete software emulation of Intel's 8086 microprocessor.
- Integrated 8086 ASSEMBLER
- Step-by-step Assembly Language Tutorials.
- Everything for coding in Assembly Language.



CODE IMPLEMENTATION

edit: C:\EMU8086\MySource\final.asm

file edit bookmarks assembler help

NEW OPEN SAVE ASSEMBLE RUN

```
001 org 100h
002
003 jmp start
004
005 msg:     db      " ********************** WELCOME TO A MINI-CALCULATOR**********************"
006 msg2:    db      0dh,0ah,"1-Add",0dh,0ah,"2-Multiply",0dh,0ah,"3-Subtract",0dh,0ah,"4-Divide", 0Dh,0Ah, '$'
007 msg3:    db      0dh,0ah,"Enter First No : $"
008 msg4:    db      0dh,0ah,"Enter Second No : $"
009 msg5:    db      0dh,0ah,"Choice Error $"
010 msg6:    db      0dh,0ah,"Result : $"
011 msg7:    db      0dh,0ah ,'********* THANK YOU FOR USING CALCULATOR *********', 0Dh,0Ah, '$'
012
013 start:  mov ah,9
014         mov dx, offset msg
015         int 21h
016         mov ah,0
017         int 16h
018         cmp al,31h
019         je Addition
020         cmp al,32h
021         je Multiply
022         cmp al,33h
023         je Subtract
024         cmp al,34h
025         je Divide
026         mov ah,09h
027         mov dx, offset msg5
028         int 21h
029         mov ah,0
030         int 16h
031         jmp start
032
033 Addition:   mov ah,09h
034             mov dx, offset msg3
035             int 21h
036             mov cx,0
037             call InputNo
038             push dx
039             mov ah,9
040             mov dx, offset msg4
041             int 21h
042             mov cx,0
043             call InputNo
044             pop bx
045             add dx,bx
046             push dx
047             mov ah,9
048             mov dx, offset msg6
049             int 21h
050             mov cx,10000
```

edit: C:\EMU8086\MySource\final.asm

file edit bookmarks assembler help

NEW OPEN SAVE

```
088
089 View:   mov ax,dx
090         mov dx,0
091         div cx
092         call ViewNo
093         mov bx,dx
094         mov dx,0
095         mov ax,cx
096         mov cx,10
097         div cx
098         mov dx,bx
099         mov cx,ax
100         cmp ax,0
101         jne View
102         ret
103
104
105 ViewNo:     push ax
106             push dx
107             mov dx,ax
108             add dl,30h
109             mov ah,2
110             int 21h
111             pop dx
112             pop ax
113             ret
114
115
116 exit:    mov dx,offset msg7
117         mov ah, 09h
118         int 21h
119
120
121         mov ah, 0
122         int 16h
123
124         ret
125
126
127 Multiply:    mov ah,09h
128             mov dx, offset msg3
129             int 21h
130             mov cx,0
131             call InputNo
132             push dx
133             mov ah,9
134             mov dx, offset msg4
135             int 21h
136             mov cx,0
137             call InputNo
```

edit: C:\EMU8086\MySource\final.asm

file edit bookmarks assembler help

NEW OPEN SAVE

```
045             add dx,bx
046             push dx
047             mov ah,9
048             mov dx, offset msg6
049             int 21h
050             mov cx,10000
051             pop dx
052             call View
053             jmp exit
054
055 InputNo:     mov ah,0
056             int 16h
057             mov dx,0
058             mov bx,1
059             cmp al,0dh
060             je FormNo
061             sub ax,30h
062             call ViewNo
063             mov ah,0
064             push ax
065             inc cx
066             jmp InputNo
067
068
069
070 FormNo:      pop ax
071             push dx
072             mul bx
073             pop dx
074             add dx,ax
075             mov ax,bx
076             mov bx,10
077             push dx
078             mul bx
079             pop dx
080             mov bx,ax
081             dec cx
082             cmp cx,0
083             jne FormNo
084             ret
085
086
087
088
089 View:   mov ax,dx
090         mov dx,0
091         div cx
092         call ViewNo
093         mov bx,dx
094         mov dx,0
```

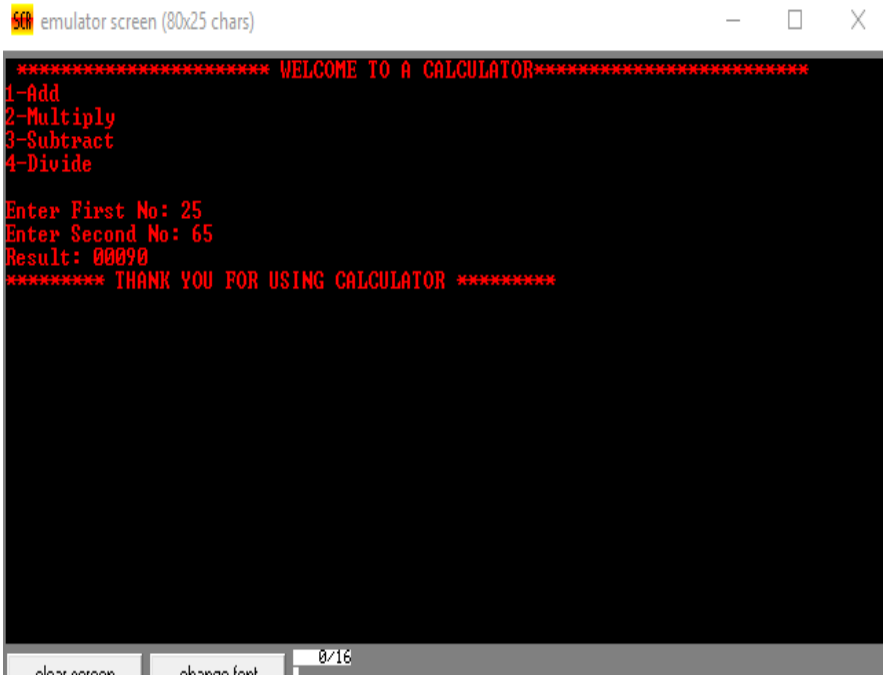
file edit bookmarks assembler help

NEW OPEN SAVE AS

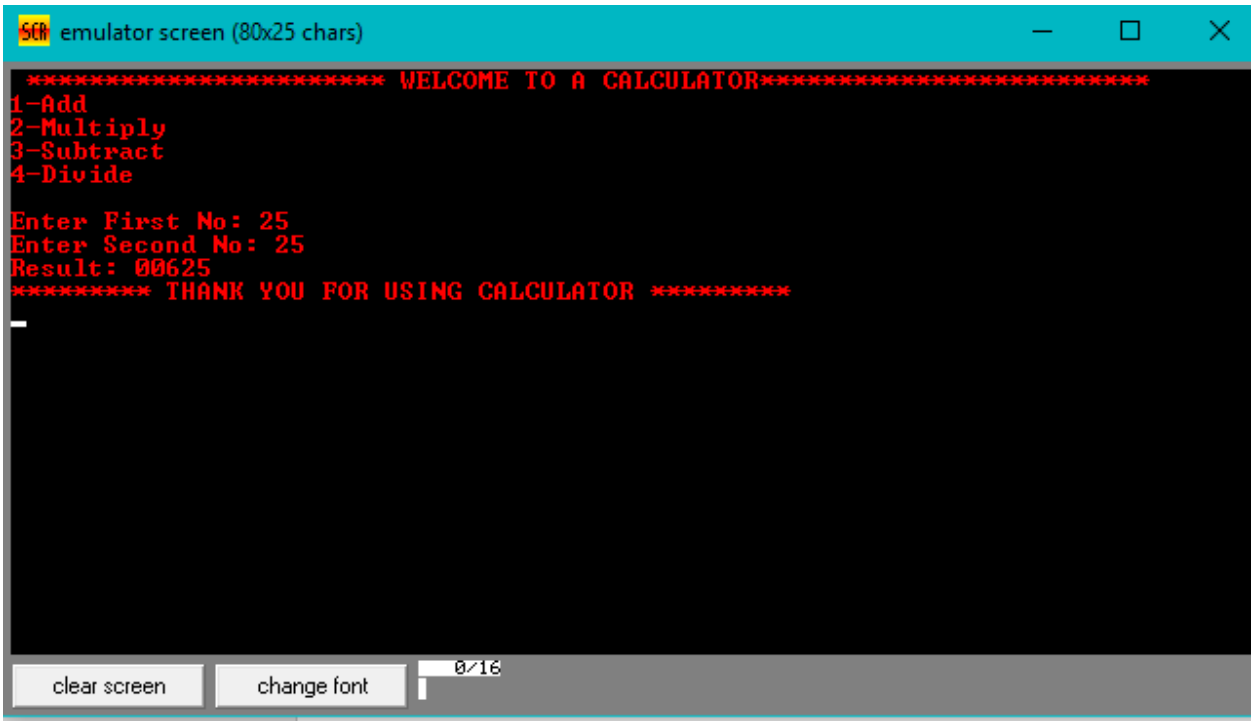
```
126
127 Multiply:   mov ah,09h
128             mov dx, offset msg3
129             int 21h
130             mov cx,0
131             call InputNo
132             push dx
133             mov ah,9
134             mov dx, offset msg4
135             int 21h
136             mov cx,0
137             call InputNo
138             pop bx
139             mov ax,dx
140             mul bx
141             mov dx,ax
142             push dx
143             mov ah,9
144             mov dx, offset msg6
145             int 21h
146             mov cx,10000
147             pop dx
148             call View
149             jmp exit
150
151
152 Subtract:   mov ah,09h
153             mov dx, offset msg3
154             int 21h
155             mov cx,0
156             call InputNo
157             push dx
158             mov ah,9
159             mov dx, offset msg4
160             int 21h
161             mov cx,0
162             call InputNo
163             pop bx
164             sub bx,dx
165             mov dx,bx
166             push dx
167             mov ah,9
168             mov dx, offset msg6
169             int 21h
170             mov cx,10000
171             pop dx
172             call View
173             jmp exit
174
175
```

```
174
175
176 Divide:     mov ah,09h
177             mov dx, offset msg3
178             int 21h
179             mov cx,0
180             call InputNo
181             push dx
182             mov ah,9
183             mov dx, offset msg4
184             int 21h
185             mov cx,0
186             call InputNo
187             pop bx
188             mov ax,bx
189             mov cx,dx
190             mov dx,0
191             mov bx,0
192             div cx
193             mov bx,dx
194             mov dx,ax
195             push bx
196             push dx
197             mov ah,9
198             mov dx, offset msg6
199             int 21h
200             mov cx,10000
201             pop dx
202             call View
203             pop bx
204             cmp bx,0
205             je exit
206             jmp exit
207
208
209
```

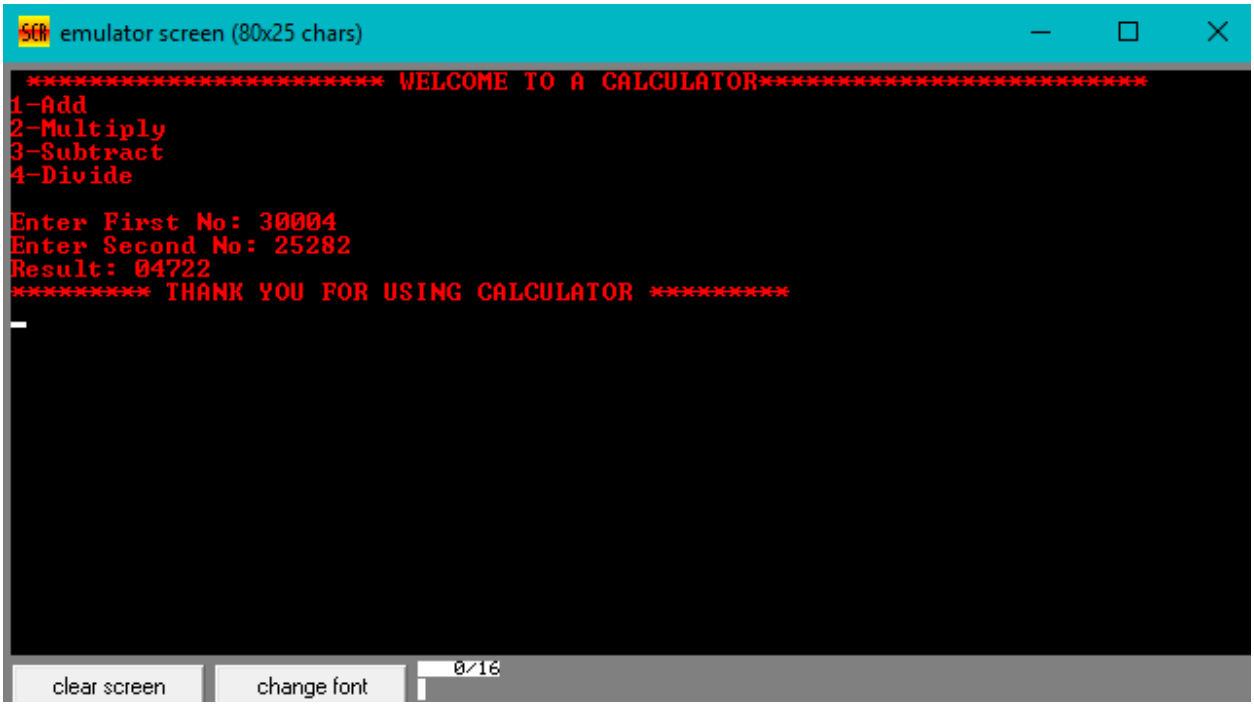
OUTPUT SCREENSHOT: - ADDITION OF TWO NUMBER'S



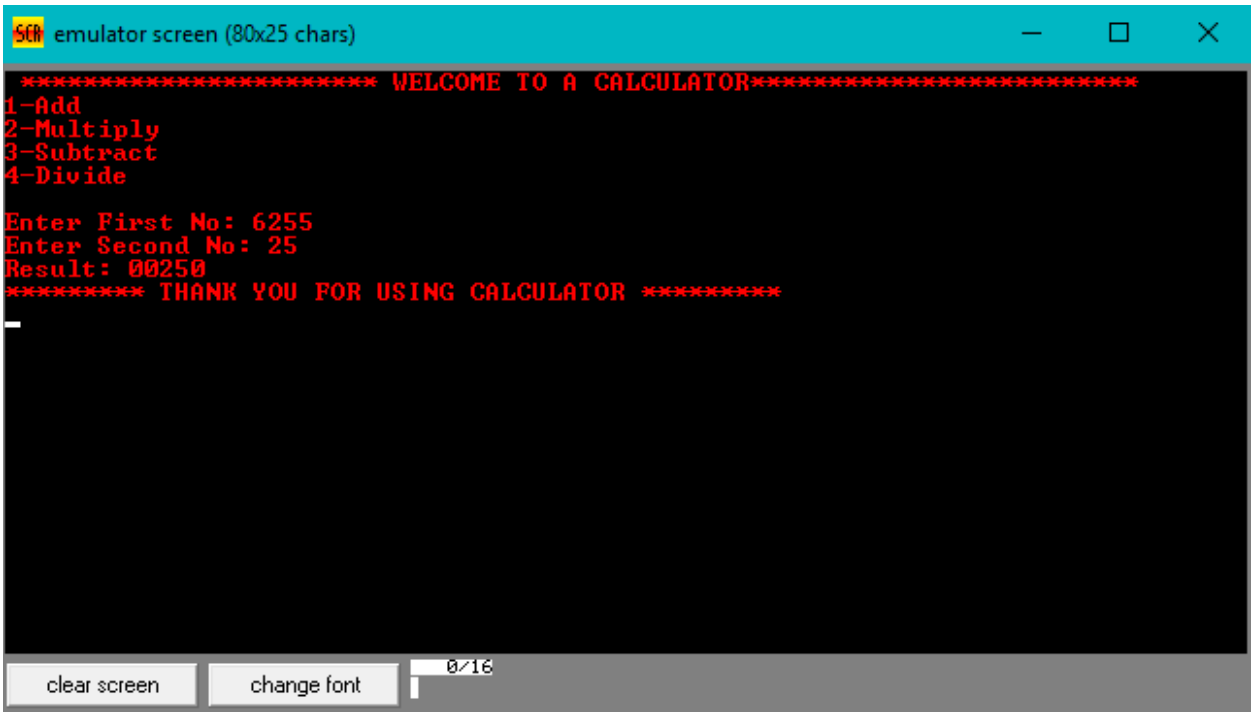
MULTIPLY:



SUBTRACTION:-



DIVISION: -



BODY MASS INDEX CALCULATION

The **body mass index** (**BMI**) is a value derived from the mass (weight) and height of an individual. The BMI is defined as the body mass divided by the square of the body height, and is universally expressed in units of kg/m^2, resulting from mass in kilograms and height in meters. The BMI is an attempt to quantify the amount of tissue mass (muscle, fat, and bone) in an individual, and then categorize that person as *underweight*, *normal weight*, *overweight*, or *obese* based on that value. Commonly accepted BMI ranges are underweight: under 18.5 kg/m^2, normal weight: 18.5 to 25, overweight: 25 to 30. To calculate your BMI: Divide your height in centimeters (cm) by your weight in kilograms (kg) The formula is as shown below: Height(cm)/Mass(kg) In our programmer we calculate our BMI applying this formula.

In this code we use some new syntax and the details is given bellows

INT 21h

Here INT 21h is used for getting input.

AND AX, 000FH

For converting the character into digit.

PUSH AX

For keeping the value of AX into the stack.

POP AX

For getting the value of AX from the stack.

MUL BX

For multiplying the value of AX with BX.

CMP AL, 0DH

For comparing the value of AL with Enter.

Here in the code we have taken the input weight(kg) and height in centimeter, the same way as calculator program I have divided the weight by height, and made the answer compare with 1,2,3,4,5.

In the standard calculator for BMI with the **metric system**, the formula for BMI is weight in kilograms divided by height in meters squared. But it is little tough to take the square of the number so I simplified the same thing by comparing it with 1,2,3,4,5.

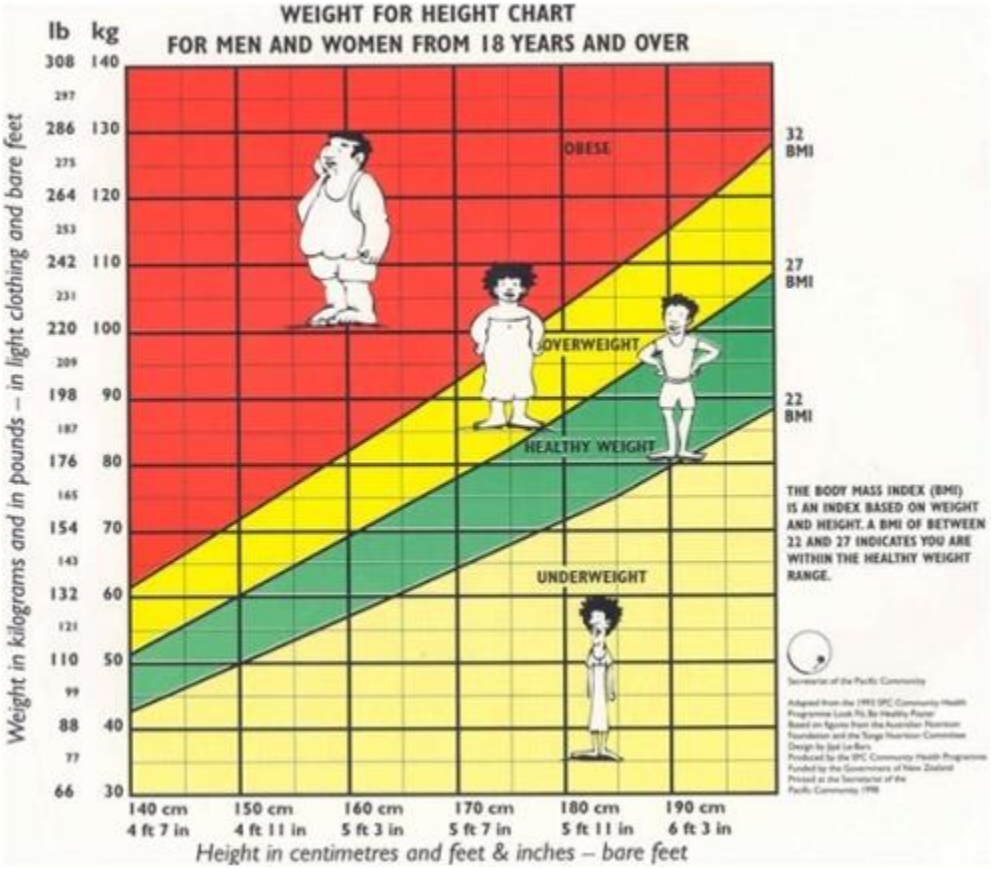
If ax=1 or 2 then it is over-weight.

If ax=4 or 5 then it is under weight.

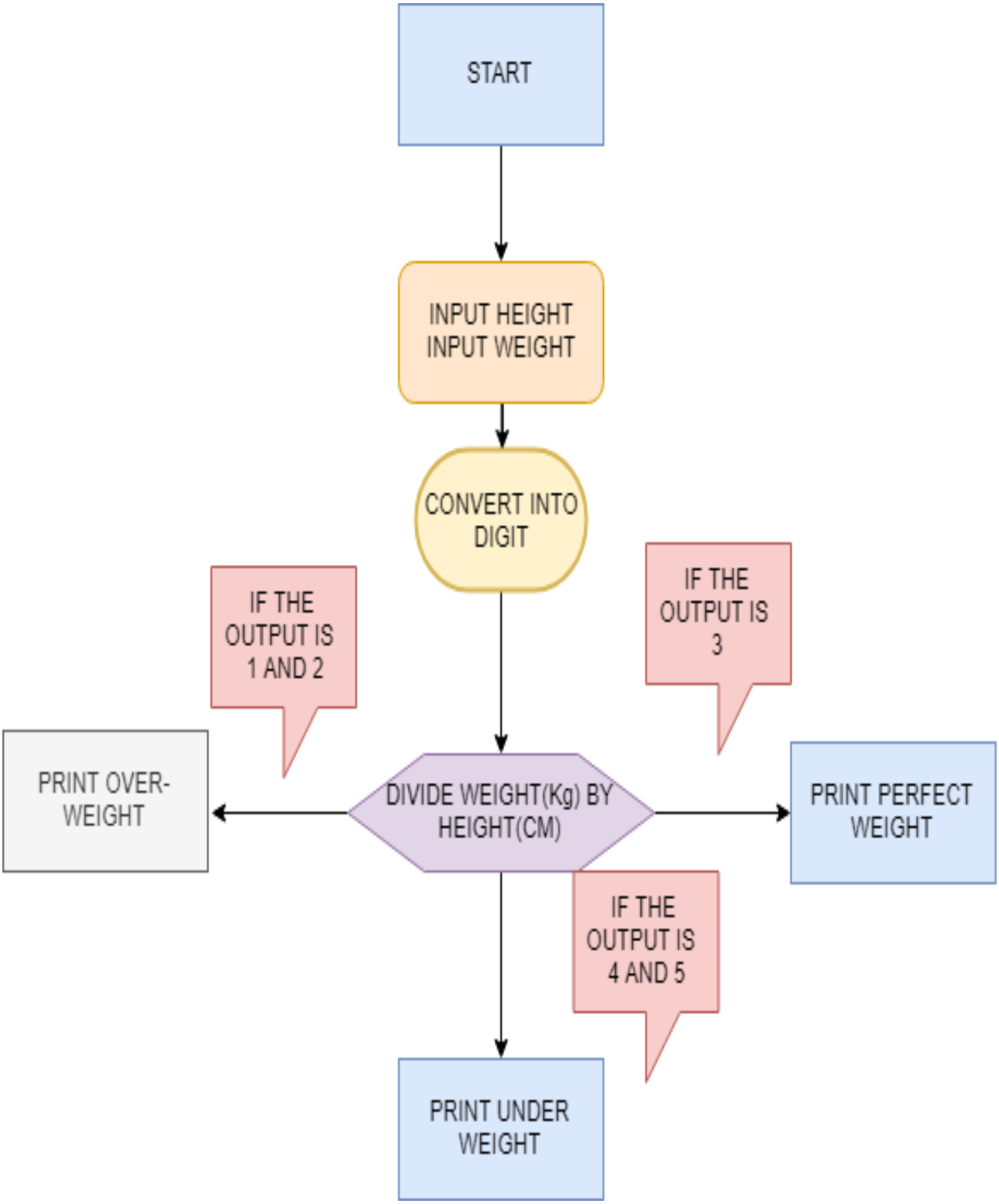
If ax=3 then it is perfect weight.

BMI is proportional to the mass and inversely proportional to the square of the height. So, if all body dimensions double, and mass scales naturally with the cube of the height, then BMI doubles instead of remaining the same. This results in taller people having a reported BMI that is uncharacteristically high, compared to their actual body fat levels. In comparison, the Ponderal index is based on the natural scaling of mass with the third power of the height.

However, many taller people are not just "scaled up" short people but tend to have narrower frames in proportion to their height. Carl Larvie has written that, "The B.M.I. tables are excellent for identifying obesity and body fat in large populations, but they are far less reliable for determining fatness in individuals.



FLOW-CHART OF CODE



CODE IMPLEMETATIO IN EMULATOR SOFTWARE-8086

edit: C:\EMU8086\MyBuild\BMI-CA.asm

file edit bookmarks assembler help

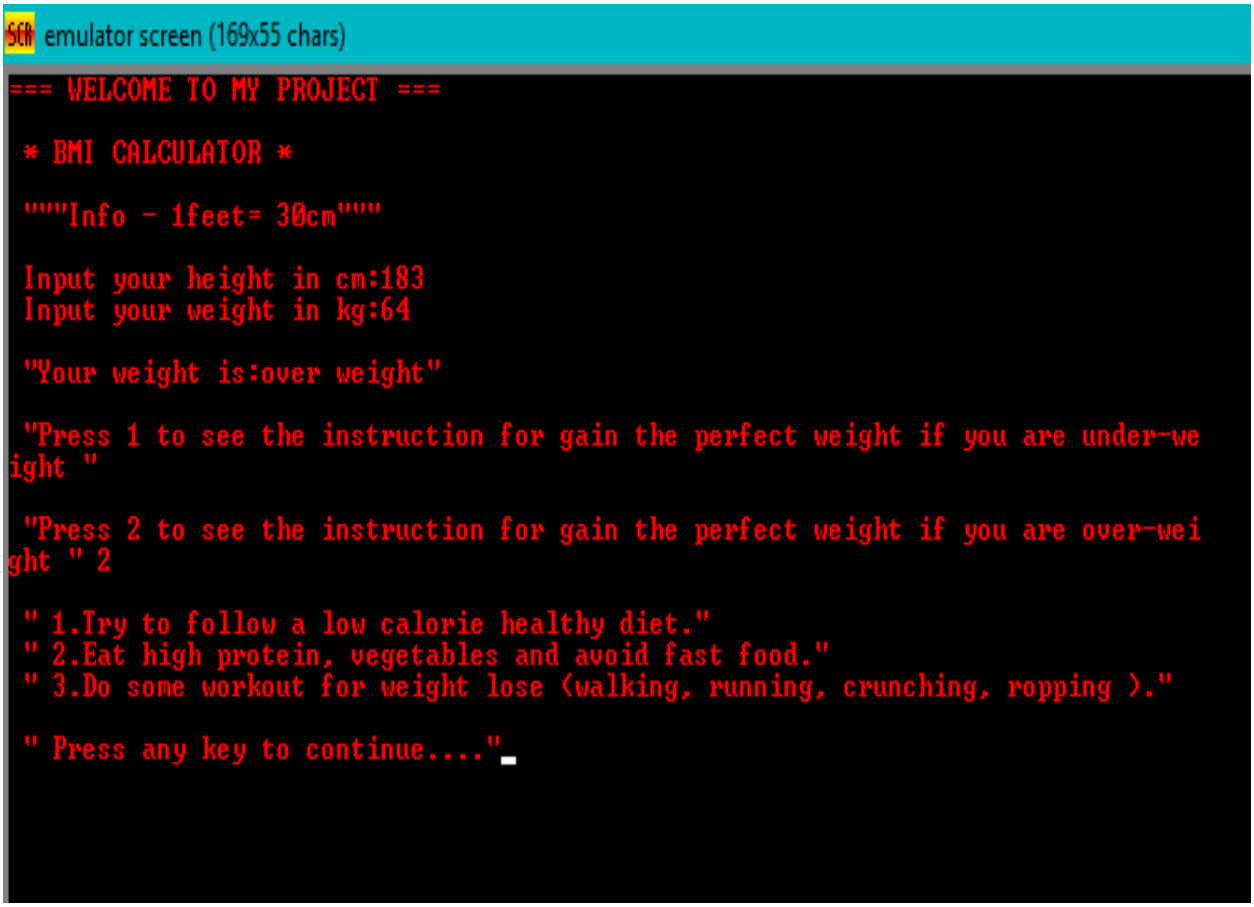
NEW OPEN SAVE ASSEMBLE RUN

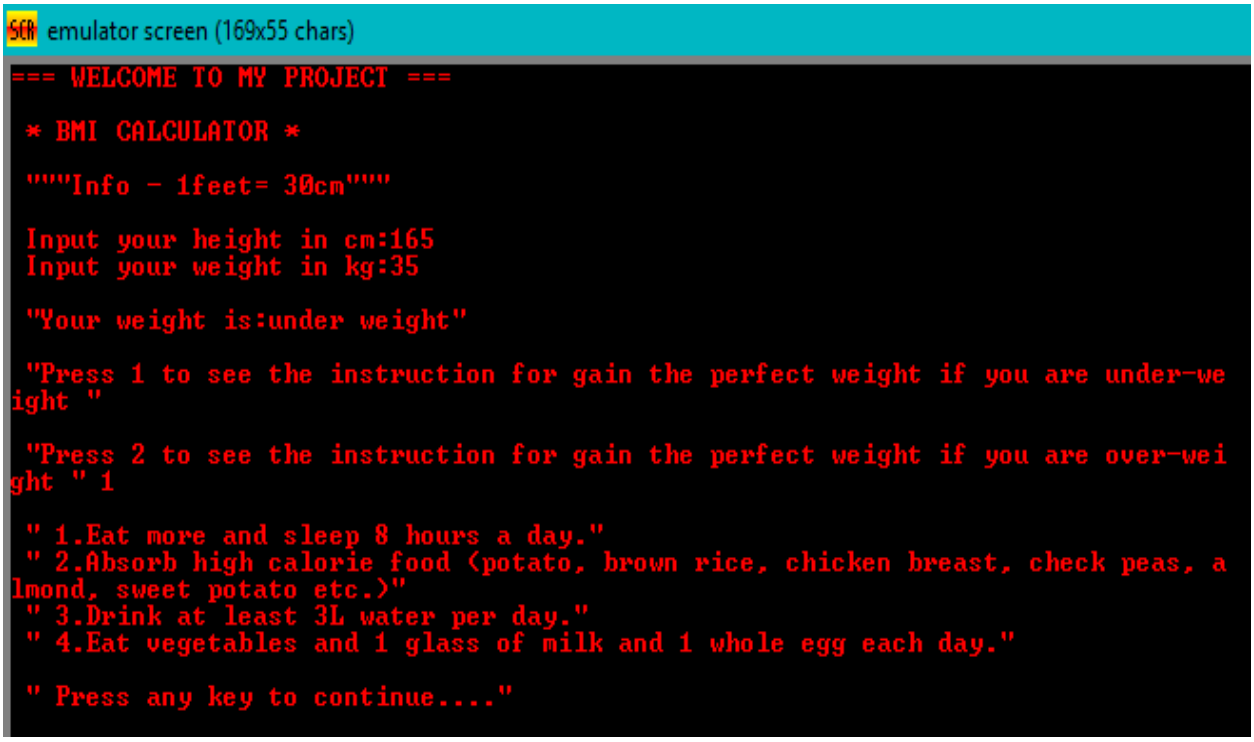
```
.MODEL SMALL
.STACK 100H
.DATA
MSA DB '=== WELCOME TO MY PROJECT ===$'
MSB DB ' * BMI CALCULATOR *$'
MSC DB ' """Info - 1feet= 30cm"""$'
MSD DB ' Input your height in cm:$'
MSE DB ' Input your weight in kg:$'
MSF DB ' "Your weight is:over weight"$'
MSG DB ' "Your weight is:perfect"$'
MSH DB ' "Your weight is:under weight"$'
MSI DB ' "Press 1 to see the instruction for gain the perfect weight if you are under-weight "$'
MSJ DB ' "Press 2 to see the instruction for gain the perfect weight if you are over-weight " $'

MSK1 DB ' " 1.Eat more and sleep 8 hours a day."$'
MSK2 DB ' " 2.Absorb high calorie food (potato, brown rice, chicken breast, check peas, almond, sweet potato etc.)"$'
MSK3 DB ' " 3.Drink at least 3L water per day."$'
MSK4 DB ' " 4.Eat vegetables and 1 glass of milk and 1 whole egg each day."$'

MSL1 DB ' " 1.Try to follow a low calorie healthy diet."$'
MSL2 DB ' " 2.Eat high protein, vegetables and avoid fast food."$'
MSL3 DB ' " 3.Do some workout for weight lose (walking, running, crunching, ropping )."$'

MSN DB ' Congratulation..! Keep it up.$'

MSM1 DB ' " Press 1 to Recalculate."$'
MSM2 DB ' " Press 2 to EXIT."$'
MSM3 DB '          ********THANK YOU********$'
MSM4 DB ' " Press any key to continue...."$'

SUM DW ?
.CODE
 MAIN PROC


  MOV AX,@DATA
  MOV DS,AX

  LEA DX,MSA
  MOV AH,9
  INT 21H

  CALL NL
  CALL NL

  LEA DX,MSB
  MOV AH,9
  INT 21H

  CALL NL
```

```
        MOV  AX,10
        MUL  BX
        MOV  BX,AX
        POP  AX
        ADD  BX,AX

        MOV  AH,1
        INT  21H

        CMP  AL,0DH
        JE  CONVERT

        JMP  INPUT2

        CONVERT:

        MOV  AX,SUM
        MOV  DX,0

        DIV  BX


        CMP  AX,1
        JE  OVER

        CMP  AX,2
        JE  OVER

        CMP  AX,3
        JE  PERFECT

        CMP  AX,4
        JE  UNDER

        CMP  AX,5
        JE  UNDER

        OVER:

        CALL  NL
        CALL  NL

        LEA  DX,MSF
        MOV  AH,9
        INT  21H

        JMP  PRESS

        PERFECT:
```

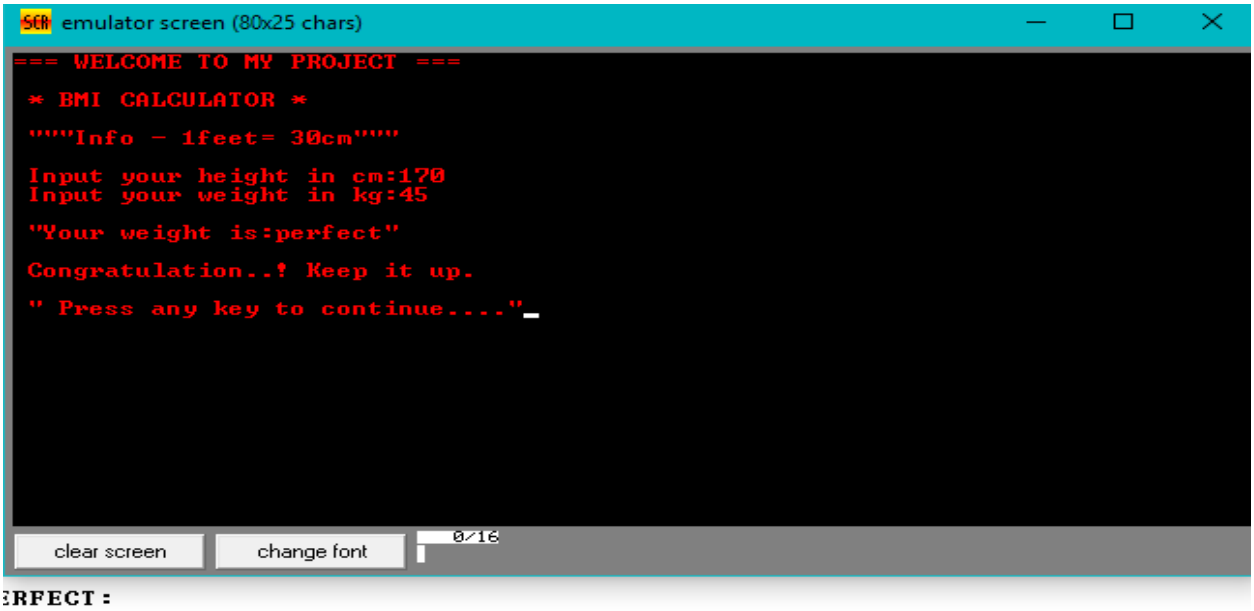
```
    CALL NL
    CALL NL

    LEA  DX,MSC
    MOV  AH,9
    INT  21H

    START:

    CALL NL
    CALL NL

    LEA  DX,MSD
    MOV  AH,9
    INT  21H

    MOV  AX,0
    MOV  BX,0
    MOV  CX,0
    MOV  DX,0
    MOV  SUM,0


INPUT:

    AND  AX,000FH
    PUSH  AX
    MOV  AX,10
    MUL  BX
    MOV  BX,AX
    POP  AX
    ADD  BX,AX

    MOV  AH,1
    INT  21H

    CMP  AL,0DH
    JE  PRINT

    JMP  INPUT

    PRINT:

    CALL NL

    LEA  DX,MSE
```

OUTPUT: - OVER WEIGHT





PERFECT WEIGHT: -



CONCLUSION

AFTER COMPLETE EXECUTION OF CODE WE WERE ABLE TO SUCCESSFULLY COMPUTE THE BASIC INSTRUCTION OF CALCULATOR LIKE ADDITION, SUBTRACTION, MULTIPLICATION, DIVISION IN x86. ALSO WE SUCCESSFULLY CALCULATED THE BODY INDEX OF HUMAN BODY USING ASSEMBLY LANGUAGE x86 WHERE WE USED THE SAME DIVISION FUNCTION WHICH WAS DERIVED FROM THE CALCULATOR PROGRAME. ASSEMBLY LANGUAGE IS NOT A DAY-TO-DAY LANGUAGE BUT A COMPUTER ENGINEERING DEVELOPER SHOULD KNOW THE ASSEMBLY LANGUAGE SO THAT BY THE PIECE OF CODE ONE CAN UNERSTAND WHAT IS GOING IN THE CENTRAL PROCESSING UNIT AND MOST IMPORTANT THING ABOUT ASSEMBLY LANGUAGE IS WHERE A PERSON WANTS TO WORK AT BYTE-BIT LEVEL.

REFERENCES

1) Assembly Language Programming Tutorial, a very thorough 55-video 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.