

**NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY**

School of Electrical Engineering and Computer Sciences

**Computer Organization & Assembly Language**

**ASSIGNMENT NO. 1,2,3**

**Title of the assignment: Creating an 8086 Simulator**

**SUBMITTED TO: Sir Imran Abeel**

**SUBMITTED BY: Mariam Babar, Qudsia Tariq, Arsalan Tariq, Aleena Ahmad, Maliha Manal**

**CLASS / SECTION: BSCS-11 / B**

**Last Date of Submission: 4th December, 2022**

# HOW TO USE:

# Copy the following files into your computer and also make sure the names of files in the last two lines of code in html file are the same as uploaded on GitHub. Then simply execute your html file on your favorite browser!

# Files to be copied:

# 8086\_simulator.html

# Simulator.js

# Simulator\_display.js

# 8086\_simulator\_graphics.css

# Happy Simulation :)

# LANGUAGES AND FRAMEWORKS USED:

1. **JavaScript:** For back-end and linking functions
2. **HTML and CSS:** For GUI development

**INSTRUCTION SET:**

We created an instruction set with the following 15 instructions and their variations:

MOV, ADD SUB, XOR, OR, AND, NOT, NEG, DIV, MUL, SHL, SHR, DEC, INC & NOP. We looked into cases of transferring of data within registers, from registers to memory and vice versa and the transfer of immediate values.

Apart from that we explored how logical operations like AND, OR, NOT & XOR worked with the storage components (registers and memory) in our simulator.

In the next section, all instructions are explained further.

**INSTRUCTIONS IMPLEMENTED**

1. **MOV**
   1. **Validity Checks:**
      * + - Value being transferred is in destination’s limit
          - Registers being invoked are included in this particular Simulator
          - Storage in Little Endian format
          - Handle both hexadecimal and decimal in Immediate
          - Decides register placement (e.g. AL,AH or AX)
          - MOV in accordance with the original 8086 instruction set
   2. **Logic Used For Implementation:**

Register and memory had different sizes of 16 and 8 bits respectively (8086 architecture), which was accommodated by using the little endian format and using the nth memory location for least significant byte and (n+1)th memory location for most significant byte. On the other hand, for transferring data from memory to register, data in the (n+1)th memory location will be stored in the H part of register, and data in nth memory location will be stored in the L part.

* 1. **Variations**
     + Mov Reg, Reg
     + Mov Reg, Immediate
     + Mov Reg, [XXXX]
     + Mov [XXXX], Reg

1. **ADD**
   1. **Validity Checks:**
      * Size of registers involved is comparable
      * Decides register placement (e.g. AL,AH or AX) in accordance with the original 8086 instruction set
   2. **Logic used for Implementation**

Conversion of hexadecimal values in registers to integers for performing addition on them and then converting them back to hexadecimal numbers so that they may be stored back into the destination register.

* 1. **Variation:**

ADD Reg, Reg

1. **SUB**
   1. **Validation Checks:**
      * Size of registers involved is comparable
      * Decides register placement (e.g. AL,AH or AX) in accordance with the original 8086 instruction set
   2. **Logic Used For Implementation:**

Conversion of hexadecimal values in registers to integers for performing subtraction on them and then converting them back to hexadecimal numbers so that they may be stored back into the destination register. Subtraction

* 1. **Variation:**

SUB Reg, Reg

1. **MUL**
   1. **Validation Checks:**
      * Decides register placement (e.g. AL or AX) in accordance with the original 8086 instruction set
   2. **Logic Used For Implementation:**

By default our multiply function only makes changes in AX register or AL depending on the size of our multiplicand, in accordance with the original 8086 MUL instruction.

Takes hex values in registers AL and the user specified register and converts them into integers. After this the integers are multiplied, and converted back to hex for storing them in AX. for a 2 byte product, lower half is stored in L register and most significant half in H register.

If two 16 bit registers are being multiplied (one of them being AX by default), the first most significant 2 bytes of the 32-bit product will now be stored in the DX register and the least significant 2 bytes will be stored in the AX register.

* 1. **Variation:**

MUL Reg

1. **DIV**
   1. **Validation Checks:**
      * Decides register placement (e.g. AL or AX) in accordance with the original 8086 instruction set
      * Cannot divide by a register holding 0000H or 00H depending on size
   2. **Logic Used For Implementation:**

If the operand given to DIV command is an 8 bit or 16-bit register, then by default the value of AL or AX will be altered respectively. Quotient will always be stored in AL register and Remainder will be concatenated into the AH register in accordance with the original 8086 instruction.

* 1. **Variation:**

DIV Reg

1. **INC**
   1. **Validation Checks:**
      * Decides register placement (e.g. AL,AH or AX) in accordance with the original 8086 instruction set
      * Cannot increment if value stored in the register is the maximum value a register can hold
   2. **Logic Used For Implementation:**

Convert hex value to integer and increase its value by 1 and then convert its base back to 16, provided that the value is not FFFFH, which is the maximum value a 16-bit register can store.

* 1. **Variation:**

INC Reg

1. **DEC**
   1. **Validation Checks:**
      * Decides register placement (e.g. AL,AH or AX) in accordance with the original 8086 instruction set
      * Cannot decrement if value stored in the register is the minimum value a register can hold.
   2. **Logic Used For Implementation:**

Convert hex value to integer and decrease its value by 1 and then convert its base back to 16, provided that the value is not 0000H, which is the minimum value a 16-bit register can store.

* 1. **Variation:**

DEC Reg

1. **NEG**
   1. **Validation Checks:**
      * Destination register is present in our map
      * Decides register placement (e.g. AL,AH or AX) in accordance with the original 8086 instruction set
   2. **Logic Used For Implementation:**

Converting the hex value in register to binary and taking its two’s complement because we have to create a negative of the value that has been passed to this function. The two’s complement function uses a for loop to flip the bits in the input string (1’s complement) and then adds 1 to the final result by using our binary addition function which ensures that the individual bits in our answer are either 0 or 1.

* 1. **Variation:**

NEG Reg

1. **AND**
   1. **Validation Checks:**
      * Value being transferred is in destination’s limit
      * AND between same sized registers
      * Registers being invoked are included in this particular Simulator
   2. **Logic Used For Implementation:**

Using a FOR LOOP, same position bits - string indexing - are compared, if both are 1, at the corresponding index 1 is stored elsewise 0 is stored.

* 1. **Variation:**

AND Reg, Reg

1. **OR**
   1. **Validation Checks:**
      * Value being transferred is in destination’s limit
      * OR between same sized registers
      * Registers being invoked are included in this particular Simulator
      * Size of value stored is equivalent to destination size (append zeros to fill remaing)
   2. **Logic Used For Implementation**

Using a FOR LOOP, same position bits - string indexing - are compared, if any is 1, at the corresponding index 1 is stored elsewise 0 is stored.

* 1. **Variation:**

OR Reg, Reg

1. **NOT**
   1. **Validation Checks:**
      * Caters to invert MSB zeros
      * Value being transferred is in destination’s limit
   2. **Logic Used For Implementation**

Using a FOR LOOP, store inverted value of at corresponding indexes.

* 1. **Variation:**

NOT Reg

1. **XOR** 
   1. **Validation Checks:**
      * Value being transferred is in destination’s limit
      * OR between same sized registers
      * Registers being invoked are included in this particular Simulator
   2. **Logic Used For Implementation**

Using a FOR LOOP, same position bits - string indexing - are compared, if both are not equal, store 1 in corresponding index, else store 0.

* 1. **Variation:**

XOR Reg, Reg

1. **SHL**

* 1. **Validation Checks:**
     + Shift should not be greater than register size
     + Set size by appending zeros by appending zeros to MSB
  2. **Logic Used For Implementation**

Use string slicing to remove the MSB to shift value, append zeros at the end.

* 1. **Variation:**

SHL Reg

1. **SHR**
   1. **Validation Checks:**
      * Shift should not be greater than register size
      * Set size by appending zeros by appending zeros to MSB
   2. **Logic Used For Implementation**

Use string slicing to remove the LSB to shift value, append zeros at the MSB.

* 1. **Variation:**

SHR Reg

1. **NOP**

**MAJOR FUNCTIONS**

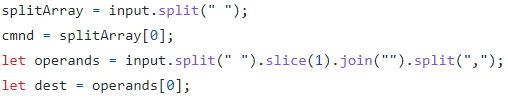
1. **PARSING**

Firstly, parsing takes the argument from ‘**submit**’ prompt in GUI, then is converts it to **all uppercase** for easy mapping in dictionaries used. Further **splitting** is done by **spaces** following by **commas** to separate the **command, destination and source**. Then according to the type of command, corresponding function of instruction is called. In case of **invalid format, error** is displayed.

Types of Errors Handled Include:

* + - Command is valid (is present in the 16 instructions we catered)
    - Register names are valid
    - Check on destination if it is memory or register
    - Source if memory, is it valid?
    - Source if immediate value, is it in range?

Split Code Snippet:



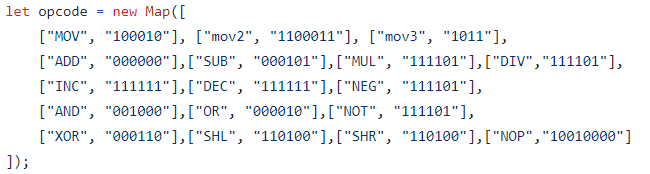
Calling of Functions Using Maps Snippet:



1. **TRANSLATION**

Corresponding to every command, destination and source, using maps the Machine Code is called. Further, mapping is called in IF STATEMENT catering for invalid instructions along with mapping the correct operands on maps. FinalCode is the variable used to store all final machine code which is achieved by concatenating multiple extracted strings from Maps.

Maps Used Snippet:



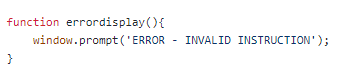
FinalCode Snippet:



**HELPER FUNCTIONS**

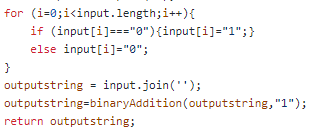
1. **ERROR DISPLAY**

Displays an error message on our GUI as soon as any instruction that violates our validation checks is submitted.



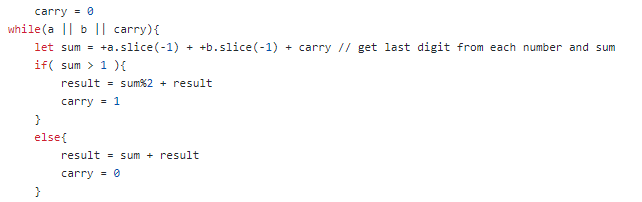
1. **TWOS COMPLIMENT**

Used by SUB & NEG instructions since subtraction or conversion of a positive number to its negative is not as simple as putting a minus sign in its beginning, our instructions used this function to find the negatives of numbers and then adding them to the other values in case of SUB.



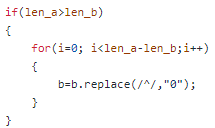
1. **BINARY ADDITION**

Ensures that addition of two binary numbers returns a sum whose individual digits are either 0 or 1.



1. **SET SIZE**

Takes two input strings of different or similar lengths and equalizes the size of both. Used in almost all instructions where some value needed to be stored in an 8-bit portion of the register or the 16-bit register itself. Function helped in converting the size of input value to the size required by its destination, so that errors may not occur.



1. **CONVERSION**

Converts numbers in a particular base, to any desired base. Used in all functions where a hex value is retrieved from registers and was converted to binary to perform a certain operation on it and then reconverted to hex for storing it back to memory or registers.



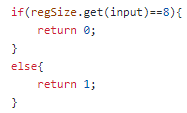
1. **RELAPCE AT**

Allows us to change the value stored at a particular index in a string. Mostly used for flipping bits where required, in instructions like NOT, etc. Takes the index and the value that should replace the current value stored at that particular index.



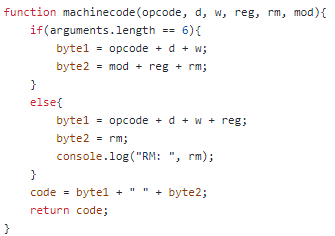
1. **SET WORD BIT**

Sets the word bit 1 if operands involved in instruction have a size of 16-bits, otherwise for 8-bit storage components, Word bit remains 0, in accordance with the 8086 architecture.



1. **MACHINE CODE**

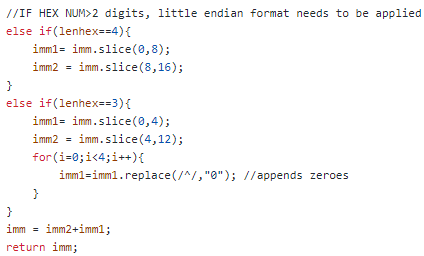
Generates a 2 byte machine code according to the original 8086 machine code, where 1st byte is assigned to opcode, direction bit and word bit; and the second byte twlls us which kind of mod our instruction is in and what operands are involved.



1. **LITTLE ENDIAN**

Register and memory had different sizes of 16 and 8 bits respectively (8086 architecture), which was accommodated by using the little endian format in which the nth memory location stores the least significant byte and (n+1)th memory location stores the most significant byte.

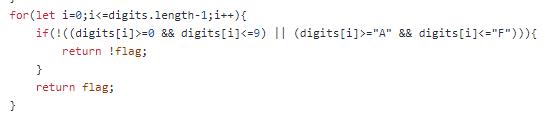
For transferring data from memory to register, data in the (n+1)th memory location will be stored in the H part of register, and data in nth memory location will be stored in the L part.



**FUNCTIONS TO CHECK THE TYPE OF OPERANDS**

1. **IS IMMEDIATE**

Checks if source is an immediate hexadecimal value which can either contain digits from 0 to 9, or alphabets from A to F. Flag is set true initially and set false on violation of checks.



1. **IS NUMBER**

Uses an in-built function to verify if our source is an immediate base 10 value, and that it doesn’t exceed the capacity of our 16-bit registers.



1. **IS MEMORY**

Returns false if a particular memory location entered by the user does not exist in our particular simulator.



**CREATING THE GUI:**

The programming languages we used were HTML and JavaScript with CSS styling (barebone).

The GUI is a cluster of <div> tags mostly with styling through css flexbox and grids. The animations are due to scaling the divs through keyframes and using setTimeout() of JS.

