آزمایش 1

اعضای گروه : مهدیه مقیسه (99243068) ، آرین جلالیان (99243026)

بخش A)

سوالات تحلیلی :

1) رجیسترهای 8088/86 در 6 دسته قرار می گیرند که برخی از آن ها 8 بیتی و برخی دیگر 16 بیتی هستند. دسته بندی آن ها به صورت زیر است :
   - رجیسترهای General که برخی از آن ها 16 بیتی هستند مانند AX و BX و برخی دیگر 8 بیتی هستند مانند AL و CH.
   - رجیسترهای Pointer که 16 بیتی هستند مانند SP و BP.
   - رجیسترهای Index که 16 بیتی هستند مانند SI و DI.
   - رجیسترهای Segment که 16 بیتی هستند مانند CS و DS.
   - رجیسترهای Instruction که 16 بیتی هستند مانند IP.
   - رجیسترهای Flag که 16 بیتی هستند مانند FR.
2) در 8086 امکان انتقال دیتا به رجیسترهایی که توسط خود پردازنده رزرو و مقداردهی می شوند، وجود ندارد. برای مثال برخی flagها و BP و IP توسط خود پردازنده مقداردهی می شوند و امکان انتقال دیتا به آن ها وجود ندارد. علت آن این است که در ساختار این پردازنده مسیر الکتریکی برای انتقال سیگنال از ورودی و خروجی حافظه به رجیسترهای segment ایجاد نشده است.
3) در میکروپروسسور 8086 سه نوع آدرس داریم که عبارتند از :
   - Physical address که با شیفت CS به اندازه یک رقم hex و اضافه کردن آن به IP (offset address) به دست می آید. مانند : 25000H+95F3H = 2E5F3H
   - Logical address که به صورت CS:IP است مانند : 2500:95F3H
   - Offset Address که همان IP است مانند : 634Ah
4) برخی رجیسترهای flag را می توان به صورت مستقیم و با استفاده از دستورات تغییر داد مانند :
   - با استفاده از دستور CLC می توان CF را صفر قرار دهد.
   - با استفاده از دستور CLD می توان DF را برابر صفر قرار داد.
   - با استفاده از دستور CMC می توان CF را برعکس کرد.

برخی دیگر از رجیسترهای flag را می توان به صورت غیر مستقیم و با استفاده از دستوراتی مانند CMP و MUL تغییر داد.

سوالات کدی:

1) در این سوال دو عدد را از کاربر می گیریم سپس مربع آن ها را حساب می کنیم و از هم کم می کنیم و حاصل را پرینت می کنیم. اگر حاصل اختلاف منفی شد صفر را به عنوان خروجی نشان می دهیم.
در بخش data متغیر هایی که خودمان تعریف کردیم را قرار می دهیم :

```
.data
    num dw 0
    result dw 0
    ten dw 10
    counter dw 0
```

در main procedure، تابع input را با استفاده از دستور call فراخوانی می کنیم و ورودی را در استک push میکنیم تا ورودی ذخیره شود و بتوانیم عدد بعدی را از کاربر بگیریم.

```
main proc near:

    call input ; input first number
    push num ; save the number
```

سپس با استفاده از دستورات زیر یک خط جدید پرینت می کنیم و در خط بعدی ورودی دوم را می گیریم :

```
    mov dx, 10
    mov ax, 200h
    int 21h ; print a new line
```

ورودی دوم را مانند ورودی اول از کاربر میگیریم. سپس اعداد گرفته شده را از استک pop میکنیم و در رجیستر ax قرار می دهیم سپس با استفاده از دستور mul، مربع آن ها را حساب می کنیم و با استفاده از دستور sub اختلاف آن ها را حساب می کنیم و در رجیستر قرار می دهیم.

```
pop ax ; move second num to ax for powering
mul ax ; ax * ax for second num

mov bx, ax ; move low part of product to bx

pop ax ; move first num to ax for powering
mul ax ; ax * ax for first num

sub ax, bx ; ax = ax - bx
```

با استفاده از دستور jnc منفی بودن یا نبودن اختلاف را بررسی می کنیم. اگر منفی نبود، به print_difference میرود :

```
jnc print_diffrence ; cf flag is set in previous instruciton
```

اگر حاصل منفی بود، تابع print را فراخوانی می کند و return می کند :

```
call print ; 0 is already in result

ret
```

Print_difference به صورت زیر تعریف شده است و پارامترها را برای استفاده تابع print در متغیر result قرار دادیم :

```
print_diffrence:

    push ax
    pop result ; result = ax which is the diffrence

    call print


ret
```

در تابع print ابتدا رجیسترهایی که در تابع از آن ها استفاده کردیم را ذخیره می کنیم و در آخر تابع آن ها را از استک restore می کنیم :

```
push ax ; store ax
push dx ; store dx

pop dx ; restore dx
pop ax ; restore ax
```

به دلیل اینکه در 8086 پرینت کردن به صورت کاراکتر به کاراکتر و با استفاده از کد اسکی آن ها انجام می شود، هر عدد رقم به رقم جدا می کنیم و کد اسکی آن را با اضافه کردن 48 به آن حساب می کنیم و آن را ذخیره می کنیم و به متغیر counter یکی اضافه می کنیم. هر بار چک می کنیم که هرگاه به صفر رسیدیم وارد قسمت بعدی می شویم :

```
start_print:

  mov dx, 0 ; empty dx
  div ten ; the remainder will be in dx

  add dx, 48 ; making the ascii code
  push dx ; save the digit
  inc counter ; increment the counter

  cmp ax, 0 ; compare the quotient to 0
  jnz start_print ; if it is not 0 repeat
```

در قسمت end_print یکی یکی رقم ها را از استک pop می کنیم و با استفاده از دستورات پرینت می کنیم :

```
end_print:

  pop dx ; pop the digit

  mov ax, 200h
  int 21h ; print

  dec counter ; decrement counter
  cmp counter, 0 ; all of the digits are printed

  jnz end_print ; repeat if counter != 0
```

در تابع input، یکی یکی رقم ها را از کاربر می گیریم و هرگاه کاربر کاراکتر enter را وارد کند به این معناست که عدد ورودی تمام شده است. ورودی گرفتن در 8086 به صورت کاراکتری است پس از هر یک از رقم ها 48 کم می کنیم تا به عدد آن برسیم و بعد عدد چند رقمی را می سازیم.

```
start_input:

    mov ah, 01h ; setting ah to input
        int     21h ; intrupt to input, result in al

        cmp al, 13 ; set zf 1 if al is an enter character
    jz end_input ; if zero flag is 1 goto end
```

```
sub al, 48 ; ascii - 48 is the actual integer
pop bx ; loading current num to bx

mov ah, 0 ; set ah to 0
push ax ; saving the digit

mov al, 10 ; move 10 to ax for mulplying
mul bx ; multiplying current num by 10 and put it in ax


pop bx ; popping saved digit back to bx

add ax, bx ; making current num

push ax ; saving new current num


jmp start_input ; back to start
```

2) در این سوال باید 6 عدد را از حافظه بخوانیم، مرتب کنیم و در حافظه بنویسیم. در قسمت دیتا یک آرایه نامرتب و همچنین سایز آرایه را قرار دادیم :

```
.data
    array dw 10h,20h,4h,8h,30h,1h  ;writing unstored numbers in array
    size dw 6
```

در تابع main با استفاده از الگوریتم insertion sort آرایه نامرتب را سورت کردیم و در حافظه مجددا قرار دادیم.
الگوریتم insertion sort به صورت زیر است :
ابتدا نقطه شروع که همان خانه دوم آرایه است را انتخاب می کنیم و اندیس آن را در cx قرار می دهیم :

```
mov cx, 2 ;setting cx to second index
```

در این الگوریتم از دو حلقه for و while استفاده می کنیم. حلقه for روی تک تک خانه های آرایه از خانه دوم حرکت می کند و در حلقه while آن را با خانه های قبلی آرایه مقایسه می کند و اگر کمتر بود جا به جا می کند.
رجیستر si اندیس آرایه برای حلقه for یعنی همان i را نگه می دارد. چون آرایه تعریف شده از نوع word است اندیس های آرایه باید هربار 2 تا جا به جا می شوند. برای اینکار هر بار از cx که همان تعداد اجرای حلقه for است یکی کم می کنیم و آن را در si قرار می دهیم سپس آن را دو برابر می کنیم و به این ترتیب اندیس آرایه به دست می آید.

```
for:
  mov si, cx ;number of comparison needed, si = cx - 1
  dec si
  add si, si
  mov ax, array[si] ;ax is key, key = array[i]
```

در حلقه while خانه آرایه که همان key است را با خانه های مقایسه می کنیم و تا وقتی که به خانه با اندیس صفر برسیم ادامه می دهیم :

```
while:
  cmp array[si - 2], ax ;if key <= array[j], exit the loop
  jbe endwhile

  mov di, array[si - 2] ;setting di to array[j]
  mov array[si], di ;array[j + 1] = array[j]

  dec si  ;next element
  dec si
  jnz while

endwhile:
  mov array[si], ax  ;key = array[j + 1]
  inc cx             ;increase i
  cmp cx, size       ;if i >= 6, exit the loop
  jbe for
```

وقتی کار سورت تمام می شود، از دستور زیر در قسمت endfor استفاده می کنیم :

```
endfor:
  mov ah, 4ch  ;exit
  int 21h
```

از آن جایی که این الگوریتم همان آرایه اولیه را مرتب می کند، پس مقادیر جدید و جا به جا شده را مجددا در همان حافظه قرار می دهد.

بخش B)

بخش ابتدایی این سوال مانند سوال 2 بخش A می باشد با این تفاوت که آرایه فیکس شده نیست و از کاربر دریافت می شود. در قسمت data متغیرهایی که در کد استفاده شده است را قرار دادیم :

```
.data
   num dw 0   ;number of elements
   result dw 0
   counter dw 0
   ten dw 10
   two db 2
   mean dw 0
   array dw 100 dup(?)



   sa db 'sorted array:', '$'
   m db 'mean:' , '$'
   mo db 'mode:' , '$'
   me db 'median:', '$'
```

ابتدا سایز آرایه را از کاربر دریافت می کنیم و آن را در num ذخیره می کنیم :

```
call input ;get number of elements
push num
mov bx, num

mov dl, 10
mov ah, 02h
int 21h ; print a new line
```

رجیستر si را اندیس آرایه و bx را سایز آرایه قرار می دهیم سپس در loop1 تابع input (که در بخش A توضیح داده شد) را فراخوانی می کنیم و با استفاده از رجیسترها ورودی ها که همان اعضای آرایه هستند را در خانه های حافظه می ریزیم. به دلیل word بودن اعضای آرایه اندیس آرایه باید 2 تا 2تا اضافه شود. همچنین برای اینکه کارمان در بخش بعدی برای محاسبه میانگین راحتتر شود هربار که ورودی میگیریم مجموع آن ها را در یک متغیر قرار می دهیم و به تعداد اعضای آرایه حلقه را پیمایش می کنیم :

```
loop1:
    call input ;get input
    push num
    pop ax
    add mean, ax  ;calculate sum of elements for mean
    mov array[si], ax ;array[i] = input
    inc si  ;i++
    inc si
    inc cx  ;counter++
    mov dx, 10
    mov ax, 200h
    int 21h ; print a new line

    cmp cx, bx ;if cx >= num, exit the loop1
    jl loop1
```

سپس تابع insertionSort (که آن را در سوال 2 بخش A توضیح دادیم) را فراخوانی می کنیم تا آرایه را مرتب کند. بعد از آن رشته sa را پرینت می کنیم :

```
    call insertionSort  ;sort the unsorted array

    mov dx,0
    lea dx,sa
    mov ah,09
    int 21h
```

loop2 را مانند loop1 پیمایش می کنیم و خانه های آرایه سورت شده را یکی یکی در متغیر result قرار می دهیم و تابع print (که در بخش A توضیح داده شده) را فراخوانی می کنیم :

```
xor si, si ;index of array, i = 0
xor cx, cx ;counter

loop2:

    mov ax, array[si] ;ax = array[i]
    push ax
    pop result
    call print ;print array[i]

    inc si  ;i++
    inc si
    inc cx
    cmp cx, bx ;if cx >= num, exit the loop1
    jl loop2
```

برای محاسبه میانگین مجموع اعداد را که در loop1 محاسبه شده است را بر سایز آرایه که در رجیستر bx قرار دارد تقسیم می کنیم و حاصل را با استفاده از تابع print پرینت می کنیم :

```
;calculate mean

mov dx, 10
mov ax, 200h
int 21h ; print a new line

lea dx,m
mov ah,09
int 21h

mov ax, mean
div bl   ;mean = sum of elements / number of elements => ax = ax / bl
mov ah, 0
push ax  ;save mean
pop result
call print ;print mean
```

برای محاسبه مد از تابع calculate_mode استفاده می کنیم و سپس با استفاده از تابع print آن را پرینت می کنیم :

```
; calculate mode

mov dx, 10
mov ax, 200h
int 21h ; print a new line

lea dx,mo
mov ah,09
int 21h

call calculate_mode
pop result ; result = mode
call print
```

در ابتدای تابع calculate_mode برای حفظ آدرس instruction تابع، متغیر address را تعریف کردیم که مطمئن باشیم با push کردن بقیه متغیرها در استک این آدرس گم نشود.

```
address dw 0

pop address ; returning address
```

از آنجایی که آرایه سورت شده است پس اعضای تکراری کنار هم قرار می گیرند. در تابع calculate_mode متغیر ctr داریم که همان counter است. همچنین یک متغیر max داریم که بزرگترین counter را نگه می دارد و متغیر mode، مد در هر لحظه را نگه می دارد. به دلیل سورت بودن آرایه از اول شروع به پیمایش آرایه می کنیم و در هر پیمایش مشخص می کنیم که چندتا از این عدد وجود دارد و اگر عدد عوض شد و counter بزرگتر از max شود ، max و mode را آپدیت می کنیم. این شرط باید برای هر عدد جدید و همچنین در انتهای حلقه for هم چک شود.

آخرین اندیس آرایه را به این صورت محاسبه و ذخیره می کنیم :

```
last dw 0 ; last index in bytes

mov ax, bx
add ax, ax
sub ax, 2
push ax
pop last ; last index in bytes = 2 * size - 2
```

متغیرها را در ابتدای کد تعریف می کنیم :

```
mode dw 0 ; will contain the mode at any stage of iteration
ctr dw 1 ; keeps the track of how many times a value is observed
max dw 0 ; max observation of an element at any stage of iteration
```

ابتدا mode را خانه اول آرایه قرار می دهیم :

```
mov si, 0 ; points to elements of array
mov ax, array[si]
push ax
pop mode ; mode = array[0]
```

سپس خانه دوم آرایه را در نظر میگیریم. سپس وارد حلقه while می شویم و در ابتدای حلقه پارامتر ها را ست می کنیم و سپس تابع is_greater را فراخوانی می کنیم تا اندیس آرایه و سایز آرایه را باهم مقایسه کند و اگر اندیس بزرگتر یا مساوی سایز آرایه بود به endwhile1 برود :

```
mov si, 2 ; point to second element

while1:

    push si ; set the parameter
    push last ; set the parameter

    call is_greater ; result will be in stack
    pop ax ; result


    sub ax, 1 ; zero flag will be 1 if pointer >= size
    jnz endwhile1
```

سپس عنصر فعلی و بعدی آرایه را با استفاده از تابع is_equal مقایسه می کنیم. اگر برابر بودند counter را یکی اضافه می کنیم و به ابتدای while بر می گردیم.

```
push array[si] ; set the parameter
push array[si - 2] ; set the parameter

call is_equal ; result will be in stack
pop ax ; result

sub ax, 1 ; zero flag will be 1 if the numbers are diffrent
jnz else
jz if


    if:
        inc ctr  ; increment ctr
        add si, 2 ; increment ctr
        jmp while1 ; go back to while
```

اگر برابر بودند، متغیرهای max و ctr را ست می کنیم و سپس با استفاده از تابع is_greater آن ها را باهم مقایسه می کنیم و اگر ctr بزرگتر از max بود، max و mode را آپدیت می کنیم. سپس اندیس آرایه را جلو میبریم و به ابتدای حلقه برمیگردیم.

```
else:
    push max ; set the parameter
    push ctr ; set the parameter

    call is_greater ; result will be in stack
    pop ax ; result

    sub ax, 1 ; zero flag will be 1 ctr is greater than max
    jz update

    push 1
    pop ctr ; ctr = 1

    add si, 2 ; increment ctr
    jmp while1 ; go back to while

        update:
            push ctr
            pop max ; max = ctr

            push array[si - 2]
            pop mode ; mode = array[i - 1]

            push 1
            pop ctr ; ctr = 1

            add si, 2 ; increment ctr
            jmp while1 ; go back to while
```

در قسمت endwhile1 متغیرهای max و ctr را ست می کنیم و با هم مقایسه می کنیم و اگر ctr از max بزرگتر بود، max و mode را آپدیت می کنیم. همچنین متغیر address را که در ابتدای تابع pop کرده بودیم را مجددا در استک push می کنیم تا بتوانیم به درستی به instruction قبلی برگردیم.

```
endwhile1:

        push max ; set the paramether
        push ctr ; set the paramether

        call is_greater ; result will be in stack
        pop ax ; result

        sub ax, 1 ; zero flag will be 1 ctr is greater than max
        jz update2

        push mode ; return mode
        push address ; returning address

        ret

        update2:
            push ctr
            pop max ; max = ctr

            push array[si - 2]
            pop mode ; mode = array[i - 1]

        push mode ; return mode
        push address ; returning address
```

تابع equal به این صورت است که دو عدد را ست می کند سپس آن ها را از هم کم می کند و اگر حاصل صفر بود، یک را برمیگرداند، در غیر این صورت صفر برمیگرداند.

```
is_equal proc near:

    ; paramathers are the two top elements in stack
    ; ( with respect to returning address )
    ; return will be 1 if the numbers are equal and
    ; will be stored in stack as well

    pop dx ; the returning address

    pop ax ; one of the numbers
    pop cx ; the other one


    sub ax, cx ; the ax will be 0 if ax and cx are equal


    jz equal ; equality


    push 0 ; return 0
    push dx ; returning address
    ret


    equal:

        push 1 ; return 1
        push dx ; returning address

    ret

is_equal endp
```

تابع is_greater هم دو عدد را ست می کند سپس آن ها را از هم کم می کند و اگر حاصل مثبت بود یک را برمیگرداند و در غیر این صورت صفر را برمیگرداند.

```
is_greater proc near:

    ; paramathers are the two top elements in stack
    ; return will be 1 if the top one is greater than
    ; the bottom one, and it will be stored in stack
    ; as well

    pop dx ; the returning address

    pop ax ; first one
    pop cx ; second one


    sub ax, cx ; ax = ax - cx will be a postive number if ax is greater than cx
    jnc greater ; cf flag is set in previous instruciton


    push 0 ; return 0
    push dx ; returning address
    ret


    greater:

        push 1 ; return 1
        push dx ; returning address
    ret


is_greater endp
```

برای محاسبه میانه در تابع calculate_median ابتدا سایز آرایه را از bx به ax منتقل می کنیم. سپس سایز را تقسیم بر 2 می کنیم تا ببینیم که تعداد اعضای آرایه زوج است یا فرد. اگر سایز آرایه فرد باشد، ah برابر 1 است و اختلاف آن ها صفر می شود و zero flag یک می شود. طبق کد اگر حاصل صفر شود به odd و اگر غیر صفر شود به even می رویم که هر کدام راه حل میانه را مشخص می کنند.

```
calculate_median proc near:

    ;pop cx ; returning address in cx

    mov ax, bx



    div two ; size / 2, remainder will be in dx
    sub ah, 1 ; zerp flag is 1 if size is odd

    jz odd
    jnz even
```

در قسمت odd، باید خانه وسط آرایه را مشخص کنیم. اندیس آن را از تقسیم قسمت قبل داریم. برای رسیدن به این خانه حافظه به دلیل word بودن اعضای آرایه باید به اندازه دو برابر اندیس آرایه جلو برویم. پس به این ترتیب خانه وسط آرایه را پیدا می کنیم، آن را در result قرار می دهیم و تابع print را فراخوانی می کنیم تا آن را پرینت کند.

```
odd:
    add al, al ; index of median in bytes
    mov ah, 0
    mov si, ax
    push array[si]
    pop result ; result is median
    call print
    ret
```

برای قسمت even، ما باید دو خانه وسط آرایه را پیدا کنیم و میانگین آن ها را به عنوان میانه پرینت کنیم. حاصل تقسیم قسمت قبل اندیس دومین خانه وسط آرایه را مشخص می کنیم. آن را در متغیر second که ابتدای کد تعریف کردیم قرار می دهیم. سپس 2 واحد از آن کم می کنیم تا به خانه قبلی آن برسیم و آن را در متغیر first که ابتدای کد تعریف کردیم قرار می دهیم. سپس با مشخص شدن محل قرارگیری این دو عنصر آرایه آن ها را در رجیستر های ax و cx ذخیره می کنیم تا بتوانیم میانگین آن ها را حساب کنیم.

```
even:
    first dw 0
    second dw 0

    add al, al ; index of second in bytes
    mov ah, 0
    push ax
    pop second ; second is index of second in bytes

    sub ax, 2 ; index of first in bytes
    push ax
    pop first ; first is index of first in bytes

    mov si, first
    push array[si] ; one of the numbers

    mov si, second
    push array[si] ; the other one of the numbers

    pop ax ; ax = second element
    pop cx ; cx = first element
```

سپس ax و cx را باهم جمع و تقسیم بر 2 می کنیم. اگر حاصل جمع آن ها زوج شود، پس عدد میانه به دست آمده صحیح است و وارد even1 می شویم و اگر حاصل جمع آن ها فرد باشد عدد میانه اعشاری است پس وارد odd1 می شویم.

```
    add ax, cx ; ax = ax + cx
    div two ;

    sub ah, 1 ; zerp flag is 1 if size is odd

    jz odd1
    jnz even1
```

در even1 برای پرینت عدد اعشاری میانه ابتدا عدد صحیح آن را که در al قرار دارد با صفر کردن ah و ذخیره ax در result پرینت می کنیم. سپس به دلیل اینکه تقسیم بر 2 شده است کافی است یک ’.5‘ بعد قسمت صحیح عدد پرینت کنیم.

```
odd1:
    mov ah, 0
    push ax
    pop result
    call print ; print the quotient

    mov ah, 2
    mov dl, '.'
    int 21h

    mov ah, 2
    mov dl, '5'
    int 21h
```

در قسمت even، کافی است ah برابر صفر قرار بگیرد و سپس ax را به result منتقل کنیم و با فراخوانی تابع print آن را پرینت می کنیم :

```
even1:
    mov ah, 0
    push ax
    pop result
    call print ; print the quotient


    ret
```