Subject

Date

آدرس دهی سیستم

```
PAG 10,150
TITE ADD 5 NUMBERS
DATA SEGMENT public PARA 'data'
DATA1 DB ?
DATA2 10H
DATA3 10111b
DATA4 -40
Sum DB ?
DATA SEG ENDS

CODESEG SEGMENT public PARA 'code)
    Assume CS:CODESEG, DS:DATASEG, ES:NOTHING, SS:NOTHING

    main : MOV Ax, DATASEG
           MOVE DS, AX

           mov AL, DATA1
           mov BL, BATA2
           ADD AL, BL
           ADD AL, DATA3
           ADD AL, DATA4
           ADD AL, DATA 5

mov Sum, AL
mov AX, 4C00H
INT 21H
CODESEG ENDS
    END main
```

سیستم عامل داشته کارش را انجام می داده تا این command را اجرا کرده حالا که کار تمام شده باید برگردد (با این دو دستور)

interrupt 21 چند service دارد که با دستور قبل از آن و move کردن در AX کاری میکنیم که با کدام service کار کند

Subject

Date

```
mov AH, 01H
INT 21H
```

منتظری باند تا از صفحه کلید چیزی را وارد کنیم
char وارد شده را در AL می‌ریزد

```
mov AH, 02H
mov DL, 30H
INT 21H
```

مرحله‌ی
byte را در داخل DL قرار گرفته به صفحه مانیتور می‌فرستد

```
mov DX, offset message
```

```
mov AH, 09H
INT 21H
```

قبل از آن باید DX را initialize کرده باشیم
message که فرستیم تا به انتهای رشته $ را چاپ کند

```
DATASEG SEGMENT Pubic PARA 'data'
    HELLO_MSG DB "HELLO WORLD $"
DATASEG ENDS
CODESEG SEGMENT Public PARA 'code'
    ASSUME CS:CODESEG, DS:DATASEG, ES:NOTHING, SS:NOTHING
main: MOV AX, DATASEG
    mov DS, AX

    mov AX, OFFSET HELLO_MSG
    mov DX, AX
    move AH, 09H
    INT 21H
CODESEG ENDS
END main
```

PAPCO

Subject

Date

تمرین — ۲۱ آبان

برنامه های زیر را با اسمبلی بنویسید که برای 5 عنصر کار کند

- Bubble Sort - Quick sort - برنامه قبلی با شیوه آدرس دهی شاخص دار

آدرس دهی غیر مستقیم

```
DATA X DB   5,15,0,21,10          در سگمنت DATASEG
SUM DB ?
CODESEG
Assume __
mov  AX, DATASEG
mov  DS, AX
```

counter نقش

```
mov  CL, 5
mov  DI, Offset DATAX            DATAX را DI بگیرید که دقیقا به خانه اش اشاره کند
mov  AL, 0  → Sum                (به جای)
Back1; ADD  AL, [DI]
     INC  DI
     DEC  CL
     JNZ  BACK1
     MOV  SUM, AL
     mov  AX, 4C00H
     INT  21
     CODESEG ENDS
     END  Main
```

موقع inc و dec کردن فقط reg را می توان قرار داد نمی شود که inc datax کرد چون که datax را point می کند که ابتدای آرایه است خودش reg نیست

Flag های نتیجه آخرین عملیات را در خودشان نگه می دارند — DEC CL اینجا

PAPCO

Subject
Date

مثال

mov CX,5

label:

Loop label = { DEC CX
JNZ label

by default برای CX کار می کند

اگر CX صفر شده باشد دیگر برنمی گردد (جهش عمل می کند فقط CX)

LoopE آدرس — LoopNE آدرس

تا زمانی ZF=0 است — ZF=0 نیست

LoopZ آدرس — Loop Z آدرس

ZF=0 نیست

باینری به BCD

اگر از 6 هر کدام رقم یک بایت BCD فشرده (placed) — 16 ← 00010000

از 6 هر رقم در 2 بایت جدا BCD غیر فشرده (unplaced) — 00000001 00000110

0000001 00000110

9 1

DATASEG { BIN_NUMBER DW 2642 H
BCD_ " DW 5DUP(?)

```
codeseg:   MOV CX,10
           mov SI,4
           mov AX,BIN_NUMBER
           DEC SI
           cmp AX,0
           JA  BACK
EXIT:      mov AX,4C00H
           INT 21H

Back: mov DX,0
      Div CX
      MOV BCD_NUMB[SI],DL
```

cmp AX,0 اگر set flag شده باشد به کمک آن تاثیر

Subject

Date

```
DATASEG SEGMENT Public PARA 'DATA'
        MESSAGE DB "Please enter a key :$"
        DATA1 DB 5
        DATA2 DB 08H
        DATA3 DB 19H
        Sum   DB ?
DATA SEG ENDS

CODE SEG SEGMENT PUBLIC PARA 'CODE'
        ASSUME CS:CODESEG, DS:DATASEG, SS:NOTHING, ES:NOTHING
        ; show message
MAIN    MOV AX, DATASEG
        MOV DS, AX
        MOV AX, CODESEG
        MOV CS, AX
        MOV AX, 0
        ADD AX, DATA1
        ADD AX, DATA2
        ADD AX, DATA3
        Printout resultt
        MOV DL, Sum
        MOV AH, 02H
        INT 21H
        MOV AX, 4C00H
        INT 21H
CODESEG ENDS
END MAIN
```

MOV AH, 09H
MOV DX, OFFSET MESSAGE
INT 21H

جمع اعداد.

کلیدی که زده میشود داخل AL به صورت byte ذخیره میشود

چاپ ASCII چاپ میکند

DATA1 به AL اضافه میشود

DATA2 به AL اضافه میشود

DATA3 به AL اضافه میشود

warning میدهد

Subject
Date

-q

-g

-e

-a

-d

تبدیل حرف کوچک به بزرگ

```
DATASEG  SEGMENT  PUBLIC  PARA  'data'
         DATA1 DB    "Enter a key"
         DATA2 DB    19 DUP(?)
         MESSAGE  DB  'Enter a key : $"
DATASEG  ENDS
CODESEG  SEGMENT  PUBLIC  PARA  'CODE'
main:    Assume
         init  Pataseg
         ; show  message  to  user
         MOV  AH, 09H
         MOV  DX, OFFSET MESSAGE
         INT  21H                    ; wait for a key
         ; MOV  AX, CODESEG
         ; MOV  CS, AX
```

-q mquite

```
         LEA  SI, DATA1
         "    , DATA2
         MOV  CS, 10
```

Subject
Date

```
Back 1: mov AL, [SI]        ← از source می‌خوانیم
        cmp AL, 61H
        JB over             ← اگر برگشتد از 61H یعنی خودش بزرگ است فقط جایش این
        cmp AL, 'Z'
        JA OVER
        AND AL, 11011111B   ← برای تبدیل کردن به حرف بزرگ
                            (استفاده از کوچک به بزرگ)
OVER: MOV [DI], AL          ← در destination می‌نویسیم
        INC SI
        INC DI              ← loop label { dec cx ; jnz cx<>0 ; Jmp label }
        loop BACK1
```

بیدا کردن بزرگترین عدد و قرار دادن آن در MAX

```
DATA1  DB  25,30,12,5,45
MAX    DB  ?
```

با برنامه debug بررسید

Subject
Date

کار با مجموعه بایت هایی که در حافظه وجود دارد:

string Instruction (آرایه ها، message ...)

REP — تکرار کردن یک دستور

REPE repeat equal — با cx کار می کنند

REPNE " not "

REPZ Repeat Zero

REPNZ " not "

(تکرار سازی)

mov CX,100 — حتما باید قبلش set cx بشه، این repeat ۱۰۰ بار تکرار می شود

REP Instruction

REPE — معمولاً مربوط به از نوع string

(Z=1 نشده باشد, CX<>0) — این repeat معادل دستور loop است.

MOVS — به DF نگاه می کند اگر یک باشد کم، اگر صفر باشد اضافه می کند

MOVSB — المان هایی که mov می شود از نوع بایت هستند (۸ بیتی) — بایت به بایت کپی می کند

MOVSW — " word (۱۶ بیتی) — دو بایت دو بایت کپی می کند

CMPS

CMPSB — مقایسه بایتی

CMPSW — " word

DF ← جهت افزایش کاهش SI و DI { ۰ ← افزایش ، ۱ ← کاهش

SCASB — داخل string بگردیم تا ببینیم word یا byte در داخل چیزی وجود دارد یا نه

SCASW

SCAS

با DI کار می کنند — چیزی که می خواهیم دنبالش بگردیم ← ۸ بیتی داخل AL ، ۱۶ " " AX

P4PCO

Subject
Date

LODS
LODSB
LODSW

یک مقدار را از SI می‌خواند و در داخل جایی که SI به آن اشاره دارد کپی می‌کند
داخل AL می‌گذارد

INS
INSB
INSW

STOSB
STOSW
STOS

مقدار AL را در جایی DI به آن اشاره دارد را در حافظه کپی می‌کند
AL در جایی که DI به آن اشاره دارد کپی می‌شود

OUTS
OUTSB
OUTSW

DI باید به destination set شود

```
SOURCE DB "string1"
DEST   DB 7Dup(?)
```

destination offset را در DI می‌ریزد

```
LED DI, DEST
LED SI, SOURCE
CLD
mov CX, 7 4
REP MOVSB   DEST, SOURCE
```

D=0 ← CLD

D=1 ← STD

CLD: DF را صفر می‌کند

حاصل source را در destination کپی می‌کند

بعد از هر بار mov کردن هم SOURCE و هم DEST یکی INC می‌شوند در MOVSB

در MOVSW بعد از هر بار 2 تا INC می‌کند

DEST

S T R I N G 1 | S | T | R | I | N | G | 1 |

```
LEA DI, DEST
LEA SI, SOURCE
CLD
mov CX, 7
REPE cmpB DEST, SOURCE
```

SOURC = "String1"

DEST = "String2"

نصف می‌شود چون 2 بایت 2 بایت counter compare می‌شود

ZF=0 و CX اگر صفر نباشد

{ CX == 0 (حتماً match شدند)
Zero flag

ربطی به ZF ندارد یعنی

یکی از این دو شرط باید با هم match نشده باشند CX≠0 و ZF<>0

Subject
Date

JCXZ match 0 — برای مشکل داشتیم این قسمت اجرا نشود

| x | CX<>0 |

JMP — اجرای شد

اگر این Jump نباشد دستورات بعدی اجرای شود باید به آخر برنامه Jump کنیم تا ببینیم دستور بعدی که باید اجرا شود چیست؟

match 0 : JZ match1

| Z<>1 |

match1:

| match اجرا شده است |

* اگر دو تا باهم یکی باشند ZF=1 می شود

message DB "String1"

```
LEA DI, MESSAGE
mov AL, 'g'
CLD
mov CX, 1
REPE SCASB message
```

همان اول ZF=1 می شود — تا به بیرون
CX تمام شده
unmatch یا به ۰ رسیده

REPNE → هنوز صفر نشده CX

extra seg

یکی از عناصر DI را می گیرد با AL compare می کند تا زمانی که نتیجه صفر شود یا CX=0 شود ZF=0 باشد

که مهم است ZF ⇔ 1 یعنی حتماً match شده حاصل compare صفر شده

JZ label1

| B | ← match این

label1

| A |

counter ZF=1 بوده

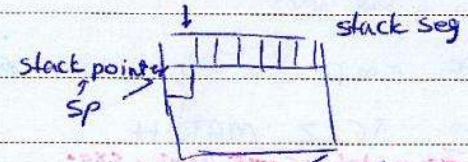
{ SI → Data seg
{ DI → Extra seg

Subject
Date

همیشه مثل از ES : DI است
DS : SI

STACKSEG s ___

X DB 64 DUP(?)



STACKSEG ENDS

باید از انتها شروع به پر کردن بکند

sp هرجا set کنیم از آنجا push میکند ، یکی یکی در دیتا کم میکند

برای اینکه در دیتا stack داشته باشیم باید یک Subcode دیگر بنویسیم

DI و ES کار میکند

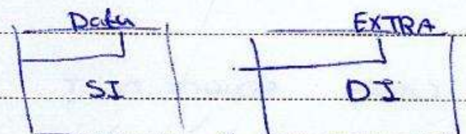
DI را داخل DEST آدرس

مثل MOV CLD

LEA DI, ES:DEST — extra segment — data در ES

LEA SI, SOURCE — داخل DS

MOV CX, 500



compare نشد پیدا بشه

REPE CMPB

همیشه DI با SI کار میکند اگر بخواهیم در دیتا String داریم SI باید SI با DI بذاریم یعنی کاری کنیم از یک یکی داشته باشند

JCXZ MATCH — CX=0 , ZF=1

DEC SI — انتها را پیدا کنیم

LOADSB SOURCE

AL ← کاراکتری که باعث match نشده است ببرد

اگر ... باشد در داخل AL میاره

MATCH:

* برنامه‌ای بنویسید که یک رشته را داخل رشته دیگر پیدا کند و اندیس اولین MATCH را برگرداند

PAPCO

SI ← LOAD

DI ← STORE

reg by default استفاده می‌شود AX یا AL است

Subject

Date

CLD → جهت ولت جهت مثبت اضافه شود

LEA SI, SOURCE

LEA DI, ES:DEST — virtual المقدار یک نوع نمایش درس

MOV CX, 100

REPE CMPSB — SI, DI را مقایسه کن تا برابر نباشند

در دستورات string همیشه با DI و SI است که seg های آنها ES و DS است

JCXZ MATCH — هر دفعه CX را کم می‌کنیم

بهم فرق نمی‌کند ... همانطور seg یکسان باشند

CX=0 و ZF=1 یعنی match هستند

DEC SI — یعنی اینکه mismatch پیش بیاید را پیدا کن

CX در counter استفاده می‌شود تأثیری در ZF ندارد

LODS SOURCE

mov SI, DI (این دستور قابل نیست)

LODS DEST → LODS همیشه با SI کار دارد برای کار با DEST باید DI را در SI بگذاریم که در اینجا SI از بین می‌رود

:

(در حاشیه:) چون mismatch پیش بیاید SI و DI یکی زیاد می‌شود و چون بعد از mismatch یکی کم می‌کنیم

MATCH :

CMPS SOURCE, DEST → نقش این در چیه؟

INS → خواندن از port

OUTS → " در port

می‌توان با رشته کار کرد یعنی به جای قرار دادن در accumulator در یک string در حافظه قرار دهیم

INS dest, DX — شماره پورت

DEST DB 10 DUP(?)

قبلش DI را باید set کرده باشیم

MOV DI, OFFSET DEST

INS DEST, DX

PAPCO

Subject
Date

دستور OUTS برعکس INS — شماره پورت

OUTS DX, SOURCE

SOURCE را به پورت می‌فرستد

highorder = 0 → CF = 0, OF = 0 — ضرب بدون علامت

CF = 0 ضرب — OF = 0 علامت

MUL → IMUL (Integer, Signed) — ضرب علامت‌دار → byte ? word ?

DIV → IDIV

بستگی به operand دوم دستور ضرب دارد
اگر ۸ بیتی → بایت
اگر ۱۶ → word

```
mov AX, 5 ;     AX = 5
mov CX, 10 ;    CX = 10
MUL CX ; {DX, AX} = AX * CX ;
```

ضرب ۸ بیتی: 8 بیت × 8 بیت → 16 بیتی → | AH (۸ بیت بالا) | AL (۸ بیت پایین) | → AL

ضرب ۱۶ بیتی: 16 × 16 → 32 → DX | ۱۶ بیت با ارزش | AX | ۱۶ بیت کم ارزش |

برای 32 بیت از دوتا ۱۶ استفاده می‌کنیم

$$\frac{Y}{X} \quad Q, \; R$$

```
mov AX, Y
mov CX, X
DIV CX
```

Q, R را generate می‌شود

Q ⇒ AL یا AX, R ⇒ DX, DL

تقسیم ۱۶ رقمی

```
mov AX, 400
mov CX, 20
DIV CX
```

DX	AX
باقیمانده	خارج قسمت

تقسیم ۸ رقمی

برای اطمینان بیشتر است AH را قبلاً صفر کنیم

```
mov AX, 40
mov CL, 5
DIV CL
```

AH	AL
باقیمانده	خارج قسمت

ادی ها با 2's complement جمع می کنند

دستور بعدی مشخص می کند این J انجام بشود یا نه

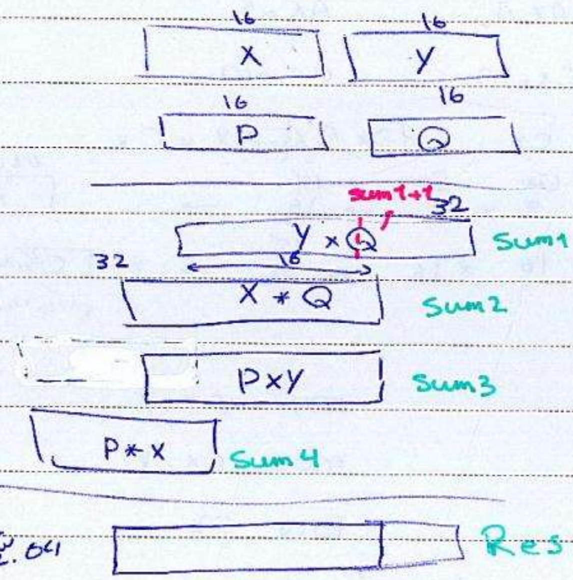
دستور را signed یا unsigned در نظر بگیریم

دید بدون علامت
```
CMP 255, 40
JA      ✓
```

دید علامت دار
```
CMP 255, 40
JG  ✗
```

0110
×101
0110
$\times 2^n$ همان شیفت به چپ
01100
011000

ضرب 32 × 32 بیتی




```
X   DW  105H          Sum1  DW  2DUP(?)
Y   DW  209H          Sum2  DW  2DUP(?)
P   DW  0405H         Sum3  DW  2DUP(?)
Q   DW  0A05EH        Sum4  DW  2DUP(?)
                      RES   DW  4DUP(?)

    MOV AX,Y
    MOV CX,Q
    MUL CX ; {DX,AX}
```

Subject
Date

```
mov  sum1, DX
mov  sum1+1*, AX

mov  Ax, x
mov  cx, Q
MUL  cx; {DX, Ax}
mov  Sum2, Dx
mov  Sum2+1, AX

mov  Ax, p
mov  cx, y
mul  cx
mov  Sum3, Dx
mov  Sum3+1, AX

mov  Ax, p
mov  cx, x
mul  cx
mov  Sum4, DX
mov  Sum4+1, AX

mov  Ax, sum1+1
mov  Res, Ax
```

Res | r | o | n | m

```
clear carry ← CLC
mov  Dx, 0
mov  Ax, sum1
mov  Bx, sum2+1
mov  cx, sum3+1
ADD  Ax, Bx
JNC  Next1
mov  Dx, 1
```

Add carry

Subject

Date

```
Next1:
        ADD  AX, CX
        JNC  Next2
        INC  DX
Next2:
        MOV  Res+1, AX
        MOV  AX, Sum2
        MOV  Bx, Sum3
        MOV  Cx, Sum4+1
        ADD  Bx, DX
        ADD  AX,BX
        MOV  DX,0
        ADD  Ax,Bx
        JNC  Next3
        MOV  DX,1
Next3:  ADD  Ax,Cx
        JNC  Next4
        INC  DX
Next4:  MOV  Res+2, AX
        MOV  Ax, Sum4
        ADD  AX, DX
        MOV  Res+3, AX
```

Subject
Date

INT ≡ CALL

INTO ≡ (overflow)

IRET ≡ return

CLD — Clear Direction flag

STD — set

CLC

STC — carry را set می‌کند (complement carry)

CMC — carry را NOT می‌کند

CLI — Clear Interrupt falg (هیچ interrupt دیگری نباشد)

STI set

فقط Interrupt هایی را می‌توانیم set یا clear کنیم که maskable (قابل جلوگیری) باشند

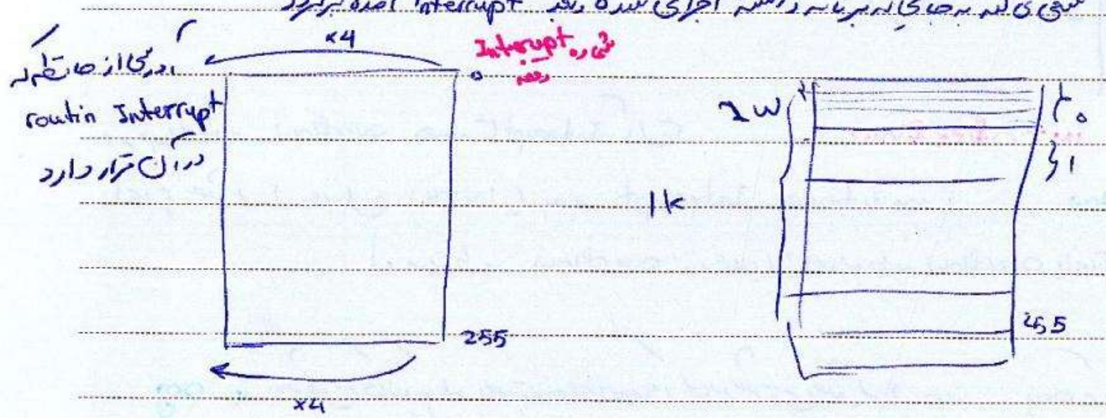
اگر بتوانیم Interrupt را فعال کنیم که maskable باشند

برای تغییر دادن بقیه flag ها از stack استفاده می‌کنیم اول در stack ، pushf می‌کنیم بعد داخل یک register می‌ریزیم و بعد از آن تغییر می‌دهیم

اگر این دستورات را برای carry flag نداشتیم با push و pop این کار را انجام می‌دهیم

```
STK SEGMENT para stack 'stack'
    X DB 128 DUP(?)
stk Ends
```

Interrupt return — سعی می‌کند به جایی که برنامه داشته اجرا می‌شده بعد Interrupt آمده برگردد



هر چهار بایت چهار بایت از اول حافظه به ترتیب شماره Interrupt است

محل خاصی که آن Interrupt اجرا می‌شود routin Interrupt

هر چهار بایت برای routin Interrupt لازم است که به صورت 2 تا 16 بیتی در نظر می‌گیریم که seg و offset است اینها را کنار هم قرار می‌دهیم می‌شود آدرس routin Interrupt ها

seg = 16 بیت اول

offset = 16 بیت دوم

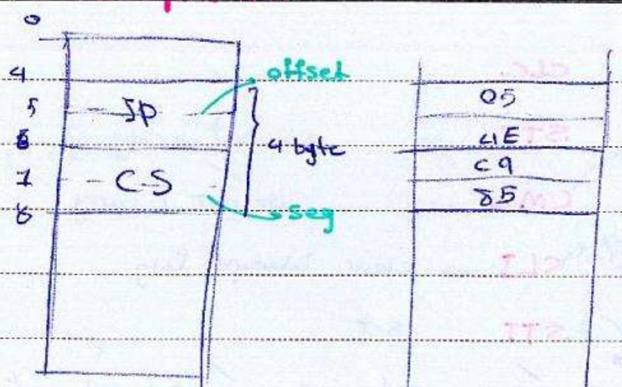
PAPCO

IP سه دستوری که می‌خواهد اجرا شود.

Subject
Date

Interrupt vector

Interrupt 7



CS:IP

برای اجرا شدن باید IP و CS به جایی که برنامه می‌خواهد اجرا شود point شود

IP 4E05

CS 85C9

۱۴ اردی

همان جایی است که Interrupt routin را نوشتیم یا نوشته شده است می‌رود و آن را اجرا می‌دهد (قبل از تغییر IP,CS آنها را در stack ، push می‌کند تا بتوانیم بعد از تمام شدن Interrupt به محل قطع شدن برنامه برگردد)

روش اجرای دستور INT
1. push کردن flag ها در stack
2. Flag مربوط به TRAPENABLE غیرفعال می‌کند
3. push کردن CS در stack
4. push کردن IP در stack
5. IP و CS را از آدرس برداری دارد

در Call کردن فقط دستورات 3 و 4 اجرا می‌شود.

روش اجرای دستور IRET →
1. pop کردن IP از stack
2. pop کردن CS از stack
3. flag [illegible]

INTO → [illegible] → در میکروپروسسور overflow Interrupt هست آن انجام شود (معادل INT4) یک conditional Interrupt

Interrupt if overflow

که مربوط به overflow که عملیات مربوط به overflow را اجرا می‌دهد

org ← مشخص می‌کنیم که دستورهای بعدی کجا نوشته می‌شود (بعد از org آدرس از حافظه قرار می‌گیرد)

2A → [illegible]
2B
2C
2D

PAPCO