Subject:
Year. Month. Date. ( )

operations

- Arithmatic operation
  - * + − / Div mod
  - SHR , SHL, ROR , ROL
    - ↳ Rotate Right
- Logical
  - and , or , XOR, NOT
- Relational ← کمتر مساوی، بزرگتر مساوی، ...
  - LE , GE , LT , LE , EQ , NE
    - ↳ not Equal

~ lable داخل این می تواند متغیر و یا label باشد

Directive

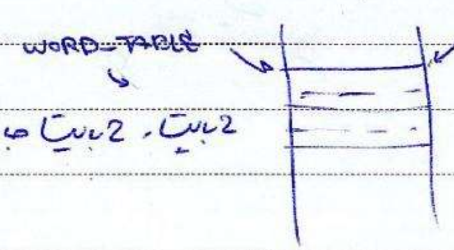
- SEG( ) ← seg آدرس را بر می گرداند بحث می شود
- OFFSET ( )
- LENGTH ( )
- TYPE ( ) → نوع
  - 1 → byte
  - 2 → word
  - 4 → Double
  - −1 → NEAR
  - −2 → FAR

A DB SEG CS:BP → آدرس را در A می ریزد

B DB TYPE A → Type A در B نوشته می شود

WORD_TABLE DW 100 DUP(?) → متغیری از نوع word که 100 خانه دارد
(DW: 2 بایتی)

FIRST_BYTE EQU PTR BYTE WORD_TABLE →
(PTR: pointer)



2 بایت، 2 بایت ها جابجا می شود

FIRST-BYTE → ابتدای word Table است ولی کاری که می کند اینه: از این طریق استفاده کنیم یک بایت یک بایت جلو می رود

PAPCO

(array)

Subject:
Year. Month. Date. ( )

PTR

SEG

OFFSET → آدرسی که مکانش به شرط offset را برمی‌گرداند

LENGTH (variable) → طول متغیر را برمی‌گرداند

far, near / word, byte, double

TYPE (variable) آ table → مشخص می‌کند از چه نوعی تعریف شده است table attribute را برمی‌گرداند

- 1 byte
- 2 word
- 4 byte
- -1 near
- -2 far

SIZE (variable) → مقدار فضایی که اشغال کرده، length * Type حاصل ضرب

مقدار بایت‌هایی که برای متغیر A استفاده شده را برمی‌گرداند

A DD 40H

MOV AX, Type A

MOV CX, SIZE A

low expression → ۸ بیت کم ارزش

High " → ۸ بیت با ارزش

asm ⟶ EXE, com

microsoft

① Masm.ExE TASM.EXE → assembler

② Link " TLink " → Linker

Turbo (borland)

EXE

③ EXE

prog.EXE prog.asm ↓ by default → prog.obj, prog.LST

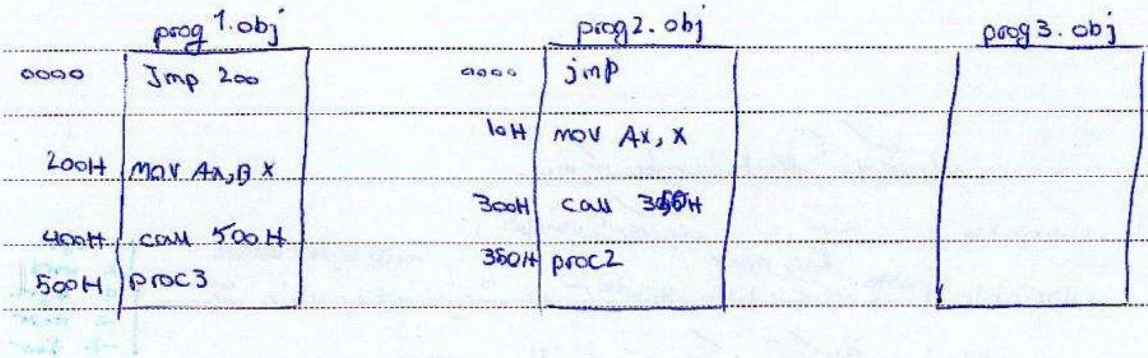
TLink.ExE → prog.obj ↓ prog.exe

prog1.asm
prog2.asm
prog3.asm
prog4.asm

prog1.asm			
jmp L4	jmp L2	jmp L3	
L4: MOV AX, BX	L2: MOV AX, X	L3: MOV AX, X	
CALL proc1	CALL proc2	CALL proc3	
proc1 proc	prog2 proc	proc3 proc	
ENDP	ENDP	ENDP	

PAPCO

Subject:
Year. Month. Date. ( )



Linker برای اینکه این سه تا به هم link کند آدرس ها را به هم می چسباند چون به ترتیب در حافظه قرار دارد
منطقه آدرس های که در برنامه دوم و سوم است باید تغییر کنند یعنی آدرس ها به اندازه برنامه قبلی link می شود
مثلا در prog2 همه آدرس ها 600 تا به آنها اضافه می شود چون
prog1 تا آدرس 600 طول دارد یعنی آدرس برنامه ای که در اول قرار گرفته
تغییر نمی کند

در کل آدرس ها تغییر کرده اند یعنی در حقیقت لازم
آدرس را linker تغییر می دهد

loader محل دقیق آدرس اجرای را مشخص می کند

PAPCO

Subject:
Year. Month. Date. ( )

com VS exe
(در حالت کلی exe راحتی است)

مزایای com نسبت به exe:
1- سرعت بالا
2- نوشتن آسان‌تر
3- سایز کم

معایب com نسبت به exe:
1- حجم کد و داده < 64k
2- DS, SS, ES → همگی یک سگمنت هستند → STACK زیرا نمی‌توانیم استفاده کنیم چون اندازه‌اش مشخص نیست ممکن است overlap پیدا کنند
3- قابلیت فراخوانی proc در seg دیگر وجود ندارد

نکات لازم برای نوشتن یک فایل com

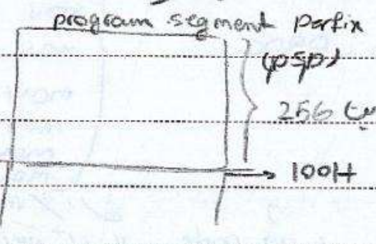
هر فایل assembly نمی‌تواند یک فایل com. generate کند

① code seg فقط (Data seg, extra seg, stack seg نداشته باشد)

② تمام proc ها از نوع Near

③ ASSUME DS:CODESEG, CS: CODESEG, ES: ~~EXTRA~~ CODESEG, SS: CODESEG
(اسم seg در تعریف آمده است) (stack را نوشتیم تا map شود) (به جای اسم همه seg‌ها اسم seg)

④- ORG 100H → برای header در ابتدا که یک قسمت اول را خالی می‌گذارد تا بیشتر شروع شود بعد از آن code seg قرار بگیرد



⑤_ END (lable) → main شروع اولین دستوری که در برنامه Run می‌شود

PAPCO

Subject:
Year. Month. Date. ( )

com vs exe — (در حالت کلی عملیات)

مزایای com نسبت به exe (است با استفاده):
1. سرعت بالا
2. نوشتن آسان‌تر
3. سایز کم

معایب com نسبت به exe:
1. حجم کد و داده < 64k
2. DS, SS, ES ← همگی یک سگمنت هستند ← STACK زیر تمام کدها است و ممکن است overlap پیدا کنند
3. در proc‌ها فقط near / seg دیگر وجود ندارد

نکات لازم برای نوشتن یک فایل com

هر فایل assembly که بتواند یک فایل com. generate کند

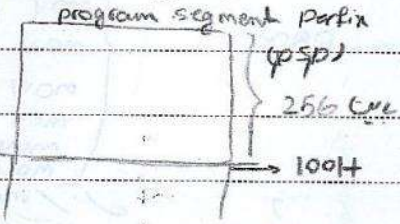
① code seg فقط (Data seg, extra seg, stack seg نداشته باشد)

② proc‌ها از نوع Near باشد

③ ASSUME DS: CODESEG, CS: CODESEG, ES: ~~EXTRA~~ CODESEG, SS: CODESEG

stack را نوشته باشیم / map. seg ... seg ...

④ – ORG 100H ← header ... ابتدای ... code seg ...



⑤ _ END (lable) ← شروع ... main ... Run می‌شود

P4PCO

Subject:

Year. Month. Date. ( )

```
TITLE PROGECT_com
CODESEG SEGMENT byte public 'code'
    ASSUME CS:CODESEG, DS:CODESEG, ES:CODESEG, SS:CODESEG
    ORG 100H
BEGIN_PROC: jmp MAIN_LABEL
    SOURCE DB 1,5,100,50
    DESTINATION DB 4DUP(?)

MAIN_LABEL: MOV AX, CODESEG
    MOV CS, AX
    MOV AX, CODESEG
    MOV DS, AX
    MOV AL, 0
    MOV DESTINATION, AL
    MOV ~ +1, AL
    ~ ~ +2, ~
    ~ ~ +3, ~

CODESEG ENDS
END Begin_PROC
```

```
MOV AL, SOURCE
MOV DESTINATION+3, AL
MOV AL, SOURCE+1
MOV DESTINATION+2, AL
MOV AL, SOURCE+2
MOV DESTINATION+1, AL
MOV AL, SOURCE+3
MOV DESTINATION, AL
```

Subject:
Year. Month. Date. ( )

RISC → Reduced Instruction Set computer

CISC → Complex " "

در معماری CISC سه نحوه دستوراتی که می‌شود از آن استفاده کرد پیچیده می‌باشد. هر یک دستور مجموعه‌ای از یک یا چند دستور ساده است.

MUL [M1], [M2], [M3] → $M_1 * M_2 = M_3$

این دستور دستور load دارد عملیات را انجام می‌دهد (ضرب) و store می‌کند

چون نسبت یک دستور چند تا عمل را انجام دهیم و با سخت‌افزار به طور راحت صحبت کنیم و مجبور نباشیم به چند دستور ساده بشکنیم. از نظر انجام، توان مصرفی، سطح ترانزیستور بزرگتر و پیچیده‌تر است

هم size هم تعداد regها بیشتر است.

RISC MOV AX, BX , store, load , MUL

↳ منطق کار کردن دستور reg است

در معماری RISC ① Instruction set

② سایز Instructionها fixed و ثابت است یعنی ۱ یا 2 بایت است

③ general purpose Reg تعداد کم است

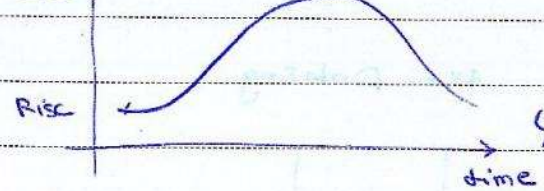
④ خود دستورات ساده و سریع الاجرا است

⑤ معماری پیاده‌سازی ساده‌ای دارد

در مقابل CISC برعکس اینها است

نمونه کامپیوترهای RISC مثل ARM، power pc، motorolla

CISC مثل Intel (80X86)، AMD، ...



در طول زمان به این نتیجه رسیده‌اند که معماری RISC خیلی بهتر است چون معماری CISC از معماری پیچیده‌ای استفاده می‌کند که باعث هدر رفت انرژی و هزینه می‌شود یعنی از 80٪ استفاده نمی‌کردند

PAPCO

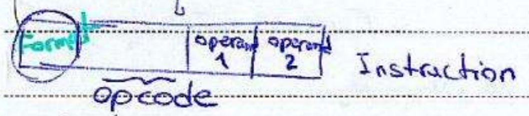
Subject:
Year. Month. Date. ( )

کاربری چندهای control است یعنی مثلاً مشخص می‌کند I/O است یا نه

اپرند ۱ و اپرند 2 → 1-2-4 بیت

بکار‌گیری مدا تبدیل می‌شود

operand 2 باید 16 بیتی باشد



opcode → تعدادی از بیت‌ها مشخص کننده opcode است یعنی خود دستور است

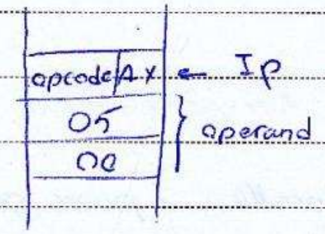
یعنی opcode مشخص داده شده همه که اپرندی خواهد بود یعنی خواهد از خودش به خودش ادامه می‌دهد و اگر نیاز باشد می‌خواند بعد از اینکه فهمید اول operand1 بعد operand2 را می‌خواند

برای خواندن operand های مورد نظر چند شیوه وجود دارد به Addressing Mode (شیوه‌های آدرس دهی)

Addressing Mode

1. Immediate Addressing (آدرس دهی بلافصل) یعنی اینکه خود عدد operand را بگذاریم یعنی عملاً مشخص کنیم و داده در خود دستور بیاید

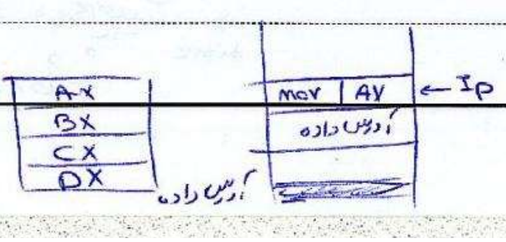
MOV AX, 05H



② Direct Addressing (آدرس دهی مستقیم) ← آدرس memory را مشخص می‌کنیم (یعنی قسمتی از حافظه مورد نظر است)

MOV AX, DATASEG   AX ← Dataseg

MOV AX, TABLE



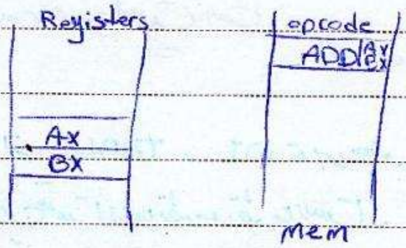
Subject:
Year. Month. Date. ( )

3_ آدرس دهی register)

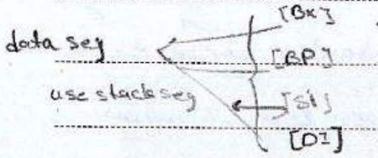
داده‌ها دیگر بری reg قرار دارند به جای کار با داده با reg کار می‌کنیم

ADD AX, BX   AX ← AX + BX

جزء Instruction نیست پس می‌دانیم چند بیت



④ آدرس دهی رجیستر غیرمستقیم (Indirect Register Addressing)

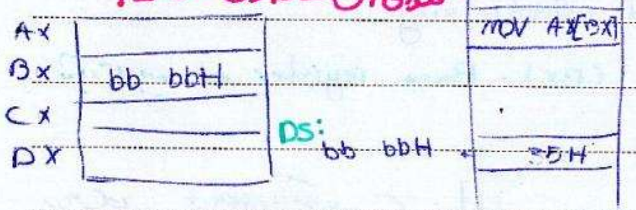


MOV AX, [BX]

[ ] منظور از [ ] آدرس از حافظه به ما بدهد

آدرس می‌بریم داده را از آن آدرس بیاوریم

BX به خانه‌ای در حافظه اشاره می‌کند که ما محتوای آن خانه را می‌خواهیم



BX در اصل آدرس offset را مشخص می‌کند

می‌توانیم حتی بگوییم offset را مشخص کنیم که بگوییم از کدام

کدام stack را می‌خواهیم CS: [BX]

base addressing mode _ [bp] , [bx]

indexed ← [SI] , [DI]

⑤ آدرس دهی نسبی با ثبات پایه (Base Register Addressing) حتی می‌توانیم ...

displacement به منظور آدرس اولش را دارد ... مشخص می‌کنیم که خانه‌ی چندم از آن

Base Relative "

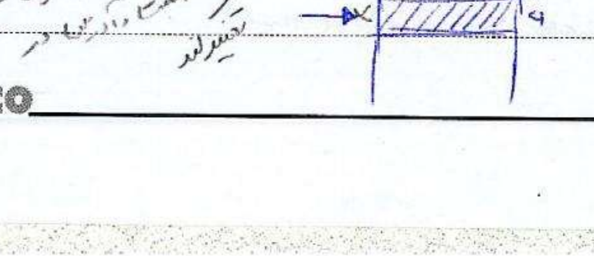
در هر دو حالت دیتا بایت را می‌خوانند

[BX]+disp

[BX+disp]

disp[BX]

داده توسط Base reg مشخص می‌شود



MOV AX, [BX]+4

۴ بیت بعد از آدرس BX را می‌خواهیم

ابتدا مشخص است و offset بعدش نیست

displacement در این حالت ثابت است ... bx می‌تواند تغییر کند

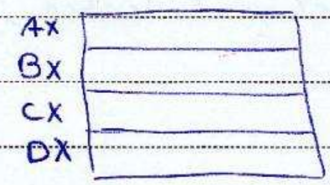
P4PCO

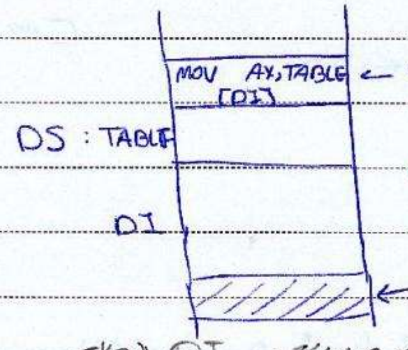
Subject:
Year. Month. Date. ( )

6. آدرس دهی مستقیم با ثبات شاخص (Direct Index Addressing)

MOV AX, TABLE [DI]

offset در حال Runtime تغییر می‌کند و در زمان در حال تغییر است / درایش شاخص نسبت است





از TABLE و DI به عنوان جابجایی استفاده می‌شود. این نحوه از آدرس‌دهی معتبر است.

آدرس اول یک data structure را مشخص می‌کنیم و DI در حال تغییر است با ثبات آدرس داده است

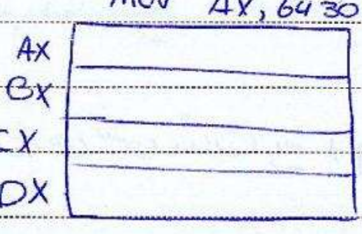
معمولا در جاهایی که از این دو استفاده می‌کنیم DS است و در 286 وقتی از DI استفاده می‌کنیم ES ممکن است باشد

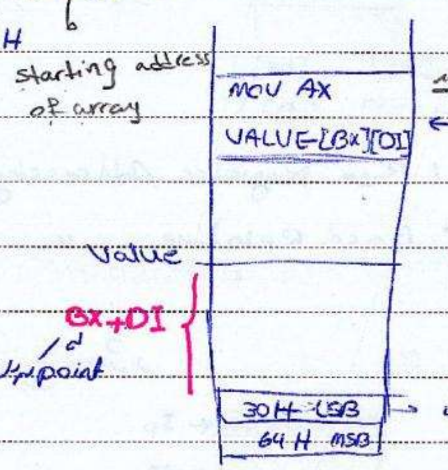
7. آدرس دهی شاخص دار با ثبات پایه
Base Indexed Addressing

ثبات پایه به Base register (BX)

MOV AX, VALUE [BX][DI]
MOV AX, 6430 H

در اینجا offset از جمع یک base reg با یک Index reg بدست می‌آید

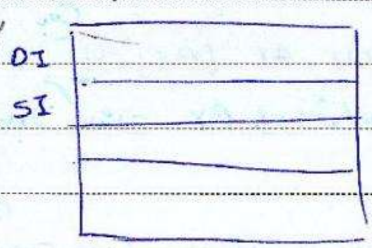




Displacement را بدست می‌آوریم از جمع DI و BX بدست می‌آید

point می‌کند به ابتدای یک record

در این حالت bp, seg stack



Runtime. value ثابت است و BX, DI در طول برنامه تغییر می‌کنند

در زبان اسمبلی operand دوم را operand می‌نامیم

P4PCO

Subject:
Year. Month. Date. ( )

Advanced Directives

Directive های پیشرفته می‌باشند

اگر در مرحله compile می‌باشد if1 یعنی pass1
if2 یعنی pass2

IF1 — IF1 کد داخل ENDF — کد را assemble می‌کند

IF2

IF NDEF — Not Define کردن

IF DEF — Define کردن — IFDEF ALT ... ENDF

#ENDF — اگر که ALT تعریف شده باشد این قسمت assemble می‌شود

IF — اگر منطقش 1 باشد اجرا می‌شود

IFIDIN

IFIDNF

IFE — اگر جلوی آن صفر باشد اجرا می‌شود در غیر اینصورت اجرا نمی‌شود

Software روی پیاده‌سازی می‌شود

استفاده از:
- Simulation شبیه‌ساز
- Emulator مقلد (تقلیدگر)

برای اجرا کردن کد یک کد اجرایی استفاده می‌کند (مثلاً مستقیم روی monitor می‌توان کار کرد)

www.emu8086.com
(crack)

Assembler ⟹ MASM.exe — اسم.asm
TASM.exe — اسم.asm

Linker — Link.exe — اسم.obj → exe
TLink.exe — اسم.obj

exe2bin.exe — اسم.exe → اسم.com

Subject:
Year. Month. Date. ( )

80X86 Instruction set

(Instruction types)

80286 ≃ 100 Instruction

چند دسته:
- نحوه اجرای برنامه (پرش‌ها) → نحوه جریان برنامه را مشخص می‌کند
- logical کارهای
- Data ضرب، تقسیم
- subroutin مربوط به

Jump →

هر وقت به دستور jump برسیم به آن آدرس می‌رود. jump بدون شرط در برنامه معنی ندارد یعنی بیکار است، مثلاً که به جای jump را بنویسیم

conditional jump:

JXX آدرس ۱۶ بیتی

هر وقت IP به این آدرس برسد، result اجرای این دستور این است که یا jump کند یا نمی‌کند

JNZ → result accu غیر صفر بشه jump کن
Jump if AX not zero

Cmp : compare

با دستور compare flag ها را set می‌کنیم و بر اساس flag ها jump می‌کنیم

cmp operand 1 , operand 2 cmp ZF =

operand 1 - operand 2 → flag ها را set می‌کند، مثل تفریق کار می‌کند

C = 1

Flags
- result = صفر ⇐ zero flag = 1
- منفی ⇐ sign flag = 0
- parity
- overflow
- carry flag

C = 1 , O = 0
C = 0 , O = 1

operand1 < operand2 → –

S=1, O=0 ;

Subject:

Year. Month. Date. ( )

اعداد منفی را با این داخل reg ذخیره می کنیم

signed → | علامت عدد two's complement → Data های مثبت یا منفی دارد

unsigned → یک عدد مثبت Data → بزرگترین Data های مثبت بیشتری را می توان ذخیره کرد که

اگر در برنامه فقط با اعداد مثبت کار کنیم پس از اول set ، unsigned می کنیم

جمع دو عدد مثبت = منفی
" " منفی = مثبت

overflow در برنامه ای که تماما اعداد مثبت باشد معنی ندارد و flag را نمی توانیم set کنیم

بخش overflow در اعداد مثبت را carry flag اجرا می کند برای اینکه تصمیم جواب درست است یا نه

cmp op1 , op2

op1		op2	
+	+	+	+
+	-	+	-
-	+	-	+
-	-	-	-
>		<	

unsigned	operand 1	operand 2	sign / relation	CF	OF	SF	ZF
=	3BH	3BH	+ = +	0	0	0	1
>	3B	15H	+ : +				
<	15	3B	+ < +				
<	F6	F9	- > -				
<	68	A5	+ > -				
>	A5	68	- < +				
>	F9	F6	- < -				

تمرین ← مقادیر جدول فوق را مشخص کنید و مشخص کنید که بزرگتر یا کوچکتر بودن.
{ (الف) برای اعداد علامت دار
(ب) برای اعداد بدون علامت

چه Flag های را set می کنند

PAPCO

Subject:
Year. Month. Date. ( )

علامت دار		بدون علامت	
3BH	0011 1011	3BH	0011 1011
− 15H	0001 0101	−15H	0001 0101

```
  0011 1011          0011 1011
+ 0001 0101          0001 0101
---------------      ---------------
(1)0010 0110         (1)0010 0110
```

CF	OF	SF	ZF		CF	OF	SF	ZF
1	0	0	0		1	0	0	0

در روش بدون علامت overflow به دردی نخورد

انواع jump

Data هایی که با آن کار می‌کنیم مثبت است

* JA : jump if above C=0, Z=0	* JNA : Jump if not above (JBE) C=1, Z=1
* JAE : jump if above or equal C=0	* JNAE : " " " " and equal
* JB : Jump if below C=1, Z=0	* JNB : " " " below and ...
* JBE : Jump if below or equal C=1, Z=1	* JNBE : " " " " and equal
	JNC : jump if not carry
JCXZ : jump if CX = 0 ZF=1, CX=0	JNG : " " " greater
JE : " " equal ZF=1	JNGE : " " " " or equal
JG : " " greater (SF=OF)	JNL : " " " less
JGE : " " greater or equal SF=OF or ZF=1	JNL : " " " " or equal
JL : " " less	JNO " " " overflow
JLE : " " less or equal	JNP : Jump if not parity

JNS : jump if not signed — JS : jump if signed

JNZ : " " " zero — JZ : " " " zero ZF=1

JO : " " overflow

JP : " " parity (EVEN) by default

JPE : jump if parity even

JPO : " " " odd

PAPCO

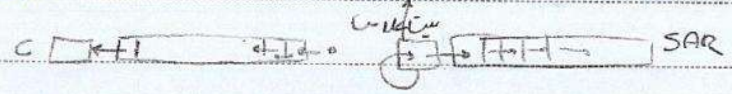
فرق JA و JG در اینست که هرکدام بر اساس یک سری از flag ها set می‌شود
همیشه ZF=0 به معنای AX=0 نیست، اگر AX تحت تاثیر قرار بگیرد ZF=0 می‌شود ولی اگر AX
تحت تاثیر قرار نگیرد نمی‌توان گفت که ZF صفر می‌شود.

تمرین: شرط های لازم برای انجام هریک از Jump ها بنویسید:

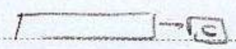
SAR shift arrithmatic Right — به سمت راست shift بدهیم بیت علامتش را تغییر نده اگر ۱ همان ۱ و اگر صفر است همان صفر بماند

SAL " " Left

SHL Shift Left

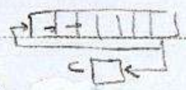


SHR " Right



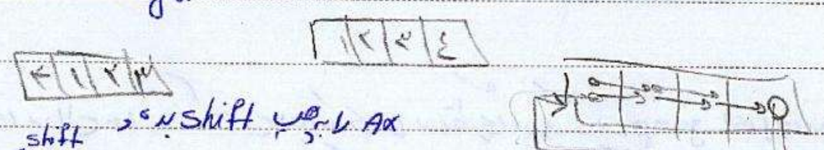
ROL Rotate Left — بیت 15 جای بیت 0 قرار بگیرد که

ROR " Right

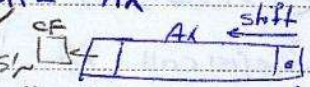


RCL Rotating with Carry Left

RCR " " " Right

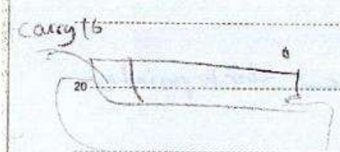


SHL AX — AX را به چپ shift بده



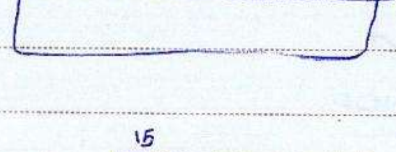
برای اینکه بیت 15 از بین نرود by default داخل carry می‌رود
در بیت صفر عدد صفر قرار می‌گیرد

SHR → by default به راست می‌رود و بیت 15 را صفر می‌گذارد



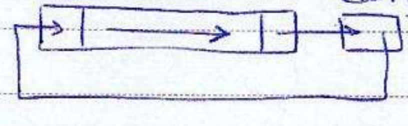
در SHR علامت sign flag تغییر می‌کند ولی در SAR تغییر نمی‌کند

RCL



carry قبلی داخل صفر می‌رود

RCR



carry قبلی را به بیت 15 می‌برد ← که قبلاً از flag ها set شده

Subject:
Year. Month. Date. ( )

تمرین) تمرین برنامه های را برای هر یک از موارد shift و rotate بنویسید که تغییرات بیت علامت و carry را چاپ کند

Jmp ~
CALL ~ اسم روال — روال اجرای برنامه را تغییر می‌دهد
RET ~ باعث می‌شود وقتی روال این کار را اجرا کند به سطر بعد از آن جایی که هست ببرد

```
SUBMOD PROC NEAR
        :
RET
SUBMOD ENDP
        :
CALL SUBMOD
```

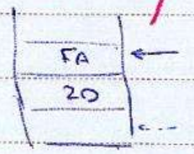
برای فراخوانی روال ها نسبت آدرس فیزیکی کار را به سطری می‌برد به جای jump اجازه می‌دهد
وقتی call اجرا می‌شود و به Return می‌رسد (به stack کار می‌کند و آدرس را پر می‌کند)
مقدار پر شده برنامه را return می‌کند

stack pointer ← از پایین کم به زیاد
pop ← مقدارش زیاد می‌شود
push ← مقدارش کم می‌شود

ret
(ENDP)

Subject:
Year. Month. Date. ( )

```
proc1                          proc2                       proc3
{                               }                            |
CALL proc2                     CALL proc3                    |
}                               ;                           ReT
proc1 ENDP                     proc2 ENDP                  proc3 ENP
```



بعد از call آدرس در stack pointer دو واحد کم می‌شود

وقتی return می‌شود pop می‌کند آدرس (call proc3) و مقدار آن را به return آدرس می‌برد

بعد از اینکه call شد آدرس دستور بعدی را نگه می‌دارد تا بعد از اجرای proc به آن برگردد در stack pointer

برای فراخوانی که می‌خواهیم به قبل برگردیم باید از stack استفاده کنیم

نحوه قرار دهی کد بین برنامه ۱) procedure را تعریف کنیم و بعد call کنیم، مثل زیر برنامه را زیاد نمی‌کند

MACRO ← directive شبیه procedure، هر وقت یک MACRO را call می‌کنیم

در جای برنامه این proc را کپی می‌کند، از stack استفاده نمی‌کند سایز برنامه

زیاد می‌شود باعث می‌شود سرعت برنامه زیاد شود

```
             MOV  AX/Offset NEXT
→ call proc1 push AX
             Jmp  proc1
20FAH
NEXT: __
```

اول آدرس برگشت مشخص می‌شود و بعد jump می‌کند

فراخوانی با stack ← چون آخرین فراخوانی باید تکلیفش مشخص شود (مورد استفاده بیشتر در recursive)

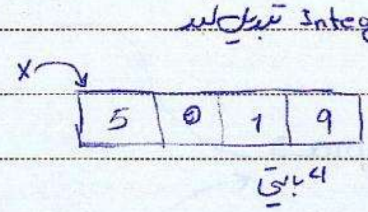
فراخوانی ← آدرس بعدی را push کنیم و بعد فراخوانی را اجرا می‌کند

یک procedure می‌تواند چند تا return داشته باشد ← چون به صورت jump عمل می‌کند

PAPCO

Subject:
Year. Month. Date. ( )

تمرین) یک برنامه‌ای بنویسید که یک عدد Integer به ASCII تبدیل کند



y = 5019

push Ax (Reg)

push Reg

push AL ← 8 بایتی را push می‌کند

push Ax → 16 بایتی را " " "

push OAEB ← دو بایتی

push CX
pop BX } ← MOV BX,CX

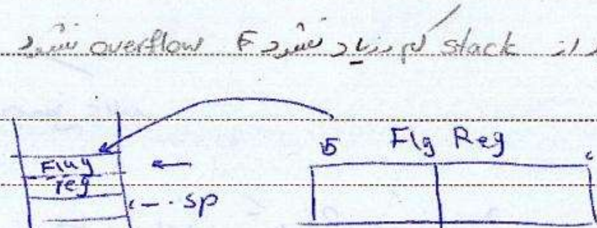
از نظر سرعت اجرا این کندتر است

بعد حالاتی پیش بیاید که از stack کم بیاید بشود F overflow بشود

از flg reg در stack پوش می‌کند

pushF

PoPF → load flag را از stack می‌خواند و می‌کند



pop A ,

push A , push All ← تمام reg های پردازنده در stack push می‌شوند

فرق stack pointer را هم در stack , push و pop می‌کند چون ... است

P4PCO

Subject:

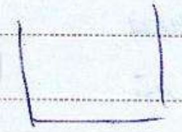
Year. Month. Date. ( )

هنگامی که sp، push می‌شود اول مقدارش همان است که در لحظه push کردن بوده ولی بعد از آن هیچ تا از آن کم می‌شود ولی اگر push ها را جدا بنویسیم مقداری که در اول push sp دارد

افزایش را می‌دهد (sp − 8)

push A

x →



push AX
" CX
" DX
" BX
" SP → stack pointer
" BD
" SI
" DI

pop A

pop DI
" SI
" BD
" SP
" BX
" DX
" CX
" AX

pushA و popA معادل با اجرای همزمان 8 push است

اگر قبل از pop کردن push نشده باشد مقادیر unknown به ما می‌دهد و این را باید خودمان Handle کنیم

pop نداشته باشیم که قبل از آن push نکرده باشیم ← مثل پرانتز بسته و باز باید push و pop با هم مساوی باشد

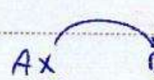
push 0
pop AX } mov AX, 0

XCHG AX, BX

بیشتر mov دو طرفه

AX هر چی دارد در BX کپی می‌شود

و BX هر چی دارد در AX کپی می‌کند



XLAT ← (pick کردن یک المان از یک آرایه)

move AL, 50 ← المان 50 اُم را اگر بخواهیم بدهیم

XLAT TABLE

AL ← TABLE [50]

قبل از استفاده از XLAT باید آدرس شروع را در BX حتما قرار دهیم

TABLE[index] ← مقدار ② Index ① AL در اینجا دو مقدار است

mov AX, BX
IN , port
OUT port

شماره پورت

INS اسم پورت، داده ← accumulator, port

OUT

داده

معادل یک آدرس مقصد است ولی از نوع (شماره پورت) و

مقصد از نوع I/O است به جای اینکه data از memory خوانده بشود از یک device خارجی خوانده می‌شود

P4PCO

معمولا شماره پورت از 0 تا 255 است و در DX می‌گذاریم

256 device

Subject:
Year. Month. Date. ( )

0 < شماره پورت < 255

immediate درصورت استفاده بصورت

IN 20H , 45H ⟶ 0045H نوشتن به دستگاه خارجی که با پورت 20H
تعریف شده است

روش کار با دستگاه‌های خارجی { OUT , IN / interrupt

شماره پورت ⇐ اگر داخل Reg قرار بگیرد
DX باشد

```
MOV DX, 20H
OUT DX, 45H
```

آدرس فیزیکی generate میشود

LEA ⟶ Load Effective address (reg16 , mem16)

LDS ⟶ Load Data Segment (pointer to) (reg16 , mem32) داده را لود میکند

LES ⟶ مثل هم هستند فقط low word را در reg که ما مشخص کردیم reg16
و high order را در DS , ES میریزد

SAHF

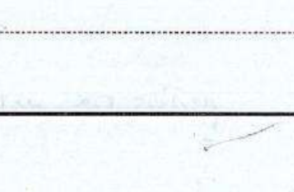
LAHF

LDS DX, TABLE[DI] { BX ← X , DS ← Y

LES DX, ____

آدرس را بدون offset ... BX میگذارد که بعد

index LEA BX, TABLE[DI] ⟶ BX
در حال تغییر است




```
MOV BX, OFFSET TABLE[DI]
ADD BX, DI
```

PAPCO