2/

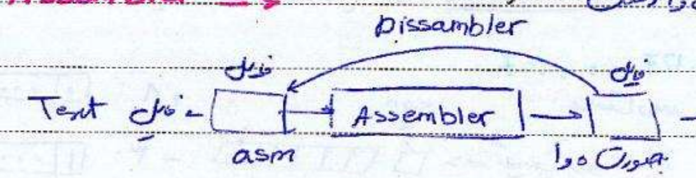
Subject:
Year. Month. Date. ( )

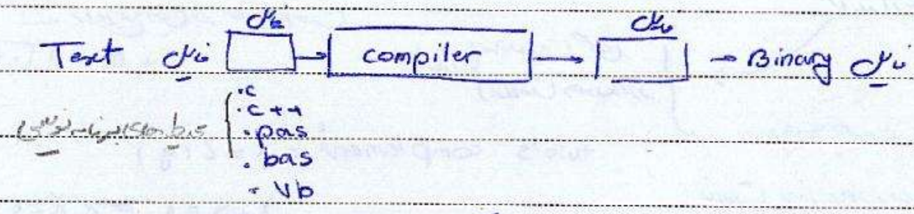
Assembler, Simulator, Disassembler, Debuger, Linker

80x86
2
3
5
4

source را به Binary تبدیل کنند

Assembler => زبان ماشین خروجی دودویی است





compiler هر یک خط دستور را بصورت دودویی تبدیل می‌شود هر یک خط دستور به خاطر abstraction بالا به چند تا دستور دودویی تبدیل می‌شود

assembler به ازای یک به یک جفت است و هر یک دستور به یک دستور دودویی تبدیل می‌شود

Dicompiler جفت نیست چون اگه اینه چند تبدیل به exe می‌شود نمی‌دانند به کدام زبان برگرداند چون هر دستور معادل چند دستور است معلوم نیست از کجا تا کجا را باید معادل یک دستور بگیرد

assembly ، compiler ندارد

Simulator

instruction ها را match نمی‌کند یک برنامه به زبان ماشین می‌گیرد و برای ما Run می‌کند

Linker

چند فایل assembly به یک exe

خروجی چند فایل assembly را با هم جمع کند و یک فایل exe تولید کند

با هم concat می‌کند

obj → exe (Linker)

L

برنامه به زبان assembly داریم که می‌خواهیم ...

... source را بنویسیم ، خطا را درس می‌...

... assembly ... ما اعداد هم جبرت ...

|...|0|1|1| = $(1\times11.10.11)_2$ ...

sign

... => □ □□□□□□□

+۸ : |0|0000000|

-۳ : |1|0000011|

که برای جمع این دو مشکل است

عدد ...

+۵ |1|0000101|

complement = X = 2(y)

$x + y = 0 \Rightarrow x = 0 - y = -y$

که در هشت بیت بیرون باشد

+ 1

=

هشت بیتی (با قاعده)

قاعده : از سمت ...

خودکار را ...

در هشت بیتی ... متقارن نیست ...

B

C

y

Subject:
Year. Month. Date. ( )

گذاری برای send و recieve استفاده می‌شود که دیگر بیت تفاوت داشته باشند (گذری)

ارزش cout یک level از cin بالاتر است

HexaDecimal — در مبنای ۱۶

1001 1101
$(9D)_{16}$ = در مبنای Hex

11111111 = −1

10000000 = −128 — عدد منفی برای این وجود ندارد

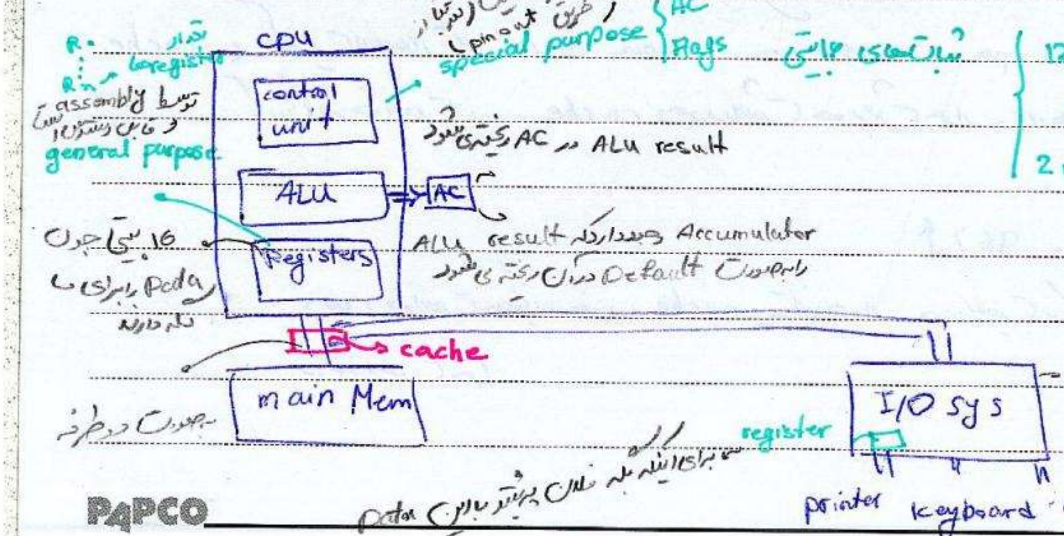
01111111 = +127

- 16 bit
- 8 bit — 8085, 8080, Z80
- 4 bit

مربوط به micropossessor، ۱۶ بیتی هست — ۱۶ بیت Data، Bass یعنی در هر لحظه از این مسیر این داده را send و recieve کنی.

80286

- Real mode Address mode — Default، مربوط به 8080 پیشتر
- protected virtual address mode
  - Data protection
  - Mem. management — بیشتر از فضای حافظه‌ای که هست استفاده کنیم به ما می‌دهد — این امکان را به ما می‌دهد



رجیسترهای ۱۴ بیتی

- 12 reg — رجیستر داده درس دهی
- 2 reg
  - Accumulator
  - Flags Register

از register برای release کردن خود استفاده می‌کنند

باعث تغییرات Flag ها می‌شود

تغییرای در AC است یک سری اثراتی دارد که در Flag ها است

→ write کردن برای special purpose ها به طور خودکار انجام می‌شود و ما نمی‌توانیم روش write کنیم

| Main Memory |

Mem Address reg (MAR): آدرس را برای main mem مشخص می‌کنیم

Mem Dataregistor ↑ (Buffer Reg — MBR, MDR): برای نگه داشتن Data که منتقل می‌شود به main Mem مسئول است که Data را نگه می‌دارد

بین CPU و main Mem یک Cache (حافظه نهان)

که تقریبا بین register و main Mem

چون Access به حافظه اکثراً تکراری هستند باعث Delay می‌شود (این Delay باعث می‌شود زمان CPU از بین برود)

on chip ← داخل CPU ← سرعت بالاتر می‌توان تکرار و هربار مسافتی را بالا ببرد

off chip ← ... CPU

cache ← آخرین اطلاعات main Mem را در خودش نگه می‌دارد، هم Data هم Address اگر آدرسی که درخواست در cache وجود داشت خودش جواب می‌دهد نه main Mem

hit ratio = 96% ↑

درصد درخواست‌هایی که خود cache می‌تواند جواب دهد نسبت به کل درخواست‌ها

Subject:
Year. Month. Date. ( )

سرعت بیشتر ↑ FF
Register
cache
Main Mem ↓ حافظه بیشتر

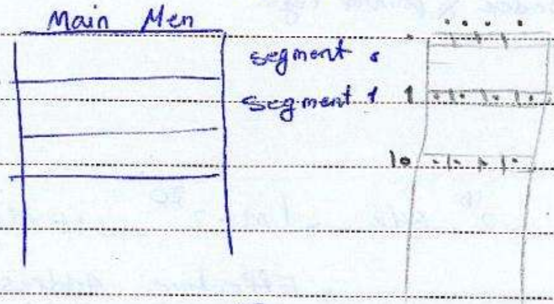
80286 آدرس دهی داده در

8085 سی 8 بیتی ← 16 بیتی ← 2^{16} آدرس مختلف = 64K

دیتا بازهم کم است در نتیجه

Segmentation ← حافظه را به قسمت های مختلف تقسیم می کنیم

هر ۱۶ بیت ۳ بیت آخر ی کنیم = segment



Segmentation سی تبدیل ۱۶ بیت آدرس به 20 بیت آدرس

هر ۱۶ بیتی که می خواهی آدرس دهی بکند تا به یک 20 بیتی تبدیل کنم

segment مشخص می کند که از ابتدای چند تا بریم جلو — offset = Address (16 بیت) ← 16 بیت آدرس

+ Segment Reg 0000

offset 15
segment reg 0000

Real Address ← Address (20 bit)
segment number و offset
19
effective address

over lap سی اگر قسمتی از segment 0 های بعد بتوانیم به استفاده دوباره segment 1 از آن استفاده کنیم

seg 4 → آدرس ؟
offset = 20

$4 \times 16 + 20 = 84$

Subject:
Year. Month. Date. ( )

دستورات به ترتیب از حافظه خوانده می‌شود و شروع به اجرا شدن می‌کند

Program counter = PC

Instruction Pointer = IP

↓

pointer از حافظه، Instruction بعدی را نشان می‌دهد

مهمترین register در cpu که آدرس دستوری که باید اجرا شود را نگه می‌دارد.

halt دستوری که به پردازنده کار را تمام می‌کند

بعد از پیچیده‌تر شدن پردازنده counter دستور به ترتیب اجرای دستور از بین رفت

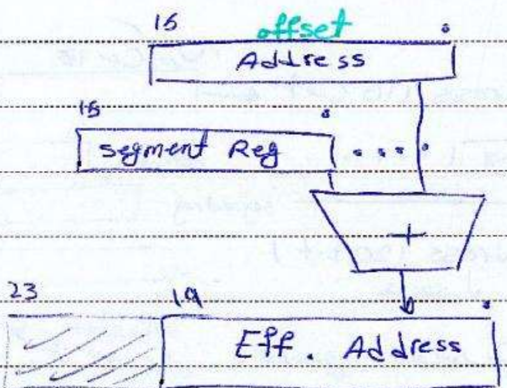
در حال حاضر مثلاً هر 5 تا دستور هر یک زمان با یک clock اختلاف اجرا می‌شوند

- 12 reg → Segment Regs, Data Regs, Index & pointer regs.
- 2 regs: AC, Flags

$16bit \rightarrow 2^{16} = 64k \rightarrow 1M = 2^{20} \rightarrow 16 Meg$

(seg, offset) ⟶ Effective Address

seg×16 + offset ⟶ physical "



بعداً اضافه شد

segment را تغییر می‌دهیم

اگر offset را صفر بذاریم آدرس‌ها مضارب segment 16 می‌شود

هر segment ، 16 بیت است؟

16 = (Seg = 0 , off = 16)

= (Seg = 1 , off = 0)

Subject:
Year. Month. Date. ( )

8088 → 64k آدرس دهی
8086
80286 → 1M آدرس دهی

با اضافه شدن ۴ بیت به 16M تبدیل شد

چهار تا از register ها مختص به segment هستند :

CS : Code Seg Reg — کد
DS : Data seg " — داده برنامه
SS : STACK " " — پشته ؟
ES : Extra " " — کارهای رشته‌ای و داده‌های زیادی را این —

برنامه به چهار قسمت مختلف تقسیم می‌شود

مثلاً ضرب و تقسیم ... result به طور Default در AX قرار می‌گیرد

high order ← H   L → low order   Accumulator

چهار تا general register
متغیرهایی که در برنامه می‌توانیم استفاده کنیم

	H	L	
به دو صورت تقسیم کرد ← words	8 bit AH byte	8 bits AL	AX
memory در Data آدرس دهی ←	BH	BL	BX Base
loop operation ←	CH	CL shift or rotate	CX count
provide port number in I/O operations ←	DH	DL	DX Data

15 0

انتهای STACK را از فضای STACK Seg استفاده می‌کند ←		SP → stack poiner
pointer آزاد که می‌شود به هر قسمت point شود ←		BP → Base pointe
موقعی لازم باشد یک sequence طبق انجام شود مثلاً برای copy کردن یک آرایه تو آرایه استفاده می‌کنیم		SI → Source Index
		DI → Destination poi Index

مربوط به برنامه هستند با اینها کار کنیم فقط ...		CS
		DS
برای اینکه تصمیم کنیم حافظه کدوم قسمتش load بشه		SS
		ES

15 0
	IP

15 0
	Flags

PAPCO

در 80286 ، clock که استفاده می‌شود معمولا 6 Meg است که کمتری می‌شود ولی بیشتر از این نمی‌تواند باشد

program speed ≃ 6 Meg $f = 6 Meg = 6 \times 10^6 \ 1/s$

$T = \frac{1}{6 \times 10^6} = 1/6 \ \mu s$

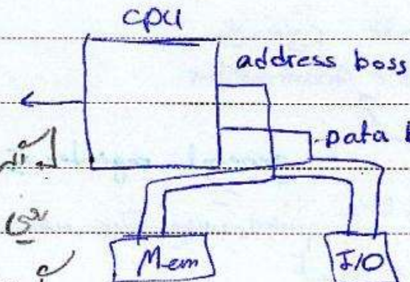
هر برنامه بستگی به اینکه چند clock مصرف می‌کند می‌توانیم زمان اجرای برنامه را محاسبه کنیم

سریع‌ترین دستور که اطلاعات را انتقال می‌دهد

MOV AX, BX → 2 clock → $\frac{1}{3} \mu s$

ADD AX, BX → x clock → $x \times \frac{1}{6} \mu s$

کار کردن با I/O نسبت به Mem کندتر است



I/M̄ ←

اگر این signal 1 باشد یعنی با IO داریم کار می‌کنیم

اگر صفر باشد با memory کار می‌کند

اگر IO/M̄ = 1 باشد address boss / Data برای IO valid است.

اگر خودمان بخواهیم IO/M̄ set می‌کنیم

اگر بخواهیم با IO کار کنیم با این دو دستور است ← Assembly { IN port#, Reg → Data را از یک Devise می‌خواند داخل این Reg بگذارد

Out port#, Reg

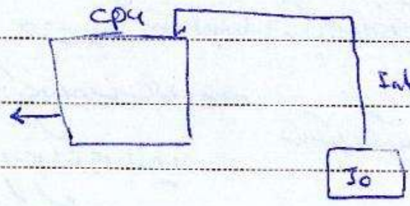
port برای کار کردن با Deviseهای IO است

مربوط به Mem. و IO/M̄=0 قرار می‌دهد

مثال → MOV AX, [20]   آدرس 20 را در AX بگذارد

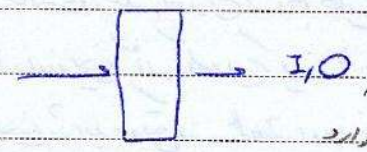
وقفه Intrupt

cpu به Data Boss و Address Boss وصل می‌کند



سیگنال‌های کنترلی { IOR / $\overline{MR}$, IOW / $\overline{MW}$

polling: سیستم یک باز مشترک بین IO و cpu



Data جهت داده‌ای این باز فرقی ندارد

cpu هر چند وقت یک بار به این بافر سر می‌زند که روش زیاد خوبی نیست چون cpu باید عملی دیگر سر بزند یا بیکار بنشیند به جای اینکه منتظر جواب IO است

از I/O device به cpu پیام Intrupt وصل می‌شود یعنی هر وقت Data I/O حاضر شد به cpu می‌گوید. cpu کارش را متوقف می‌کند به سراغ I/O می‌رود، Interupt مشخص می‌کند cpu باید از کجا شروع کند و کار عملی را انجام بدهد

hardware] HW
software] SW

هر Intrupt را Insiate می‌شود یا سخت افزاری یا از طریق برنامه‌ای که می‌نویسیم

Intrupt No : شماره وقفه

Intrupt vector

Int routine در حافظه نوشته شده است. که هر Intrupt No یک procedure دارد

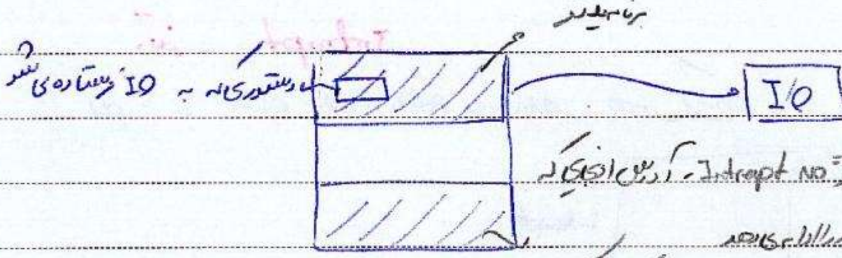
از یه جایی به بعد Intrupt routine آزاد است در دسترس کاربر است

Intrupt No.

آدرس شروع Intrupt ما

Address of Interupt

Subject:
Year. Month. Date. ( )



اگه در حال اجرای برنامه اول به IO برود
شاید یک برنامه اجرا نشده به اجرای برنامه
نفر بعدی برود تا زمانی که جواب IO بیاید، Intrept NO = 6، آدرس اونجایی که
برنامه اول متوقف شده داره و یکی از زیر برنامه‌ها را اجرا می‌کند
در هر جا که متوقف شود آدرس اونجا رو نگه داریم
برود و وقتی جواب return کنه به اونجا برگرده

هر وقتی که دستور از I/O برسد کنترل از آنجا خارج می‌شود
Interupt اولویت دارند یعنی اگر الان یک Int برسد نگاه می‌کنه که هر کدوم که اولویت بیشتری داشت
اونرا ادامه می‌دهد. یعنی شاید موقعی که Int اومد تا دستش خودش release نکنه

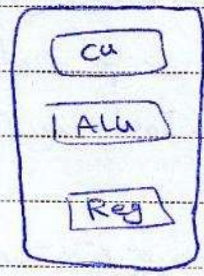
دقت Int اسم در اولین جایی که بتواند کارش را stop می‌کند و به سراغ Int می‌رود.
در 80286 حدود 256 تا Interupt داریم که از ۰ تا 32 مربوط به پردازنده است.

Int 0 : Divid by Zero هر وقت این اتفاق بیفتد پردازنده یک Int برای خودش می‌فرستد
Int 1 : single step هر دستوری که اجرا شد signal میده که دستور اجرا شد و
دستور بعدی را بخواهد ← step by step ایجاد می‌شود

reserve شده یعنی یا جایش تو حافظه reserve شده و یا Task اون مشخصه
که می‌توانیم خود Task را تغییر بدهیم

PAPCO

MIMD → Multi Inst, Multi Data



حافظه مشترک بین چند پردازنده

یعنی برای حافظه‌های با IO زیاد کار

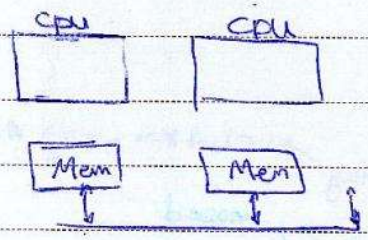
دارد مناسب نیست

قدرت پردازش بالا، کمترین دسترسی به حافظه

برای کار IO زیاد به حافظه خوب نیست برای اینکه باید fetch بعد process شود

مناسب است مثلا بعد [illegible]

Multi processor ← (Multi computer)



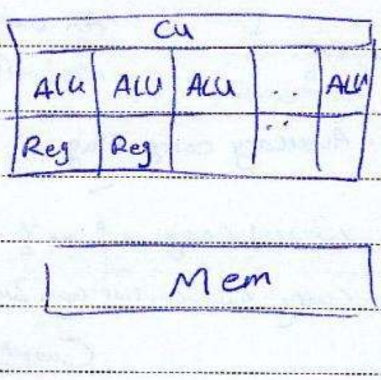
هرکدام CPU های دارند مثل اینکه چندتا کامپیوتر را شبکه کرده باشیم

یکی سرعتش بالاتر باشد چون که حافظه local دارند

مثل اینکه چند تا PC را شبکه کرده باشیم که با سرعت بیشتر اطلاعات

local خود را پاس می‌دهند برای اینکه پردازش‌ها)

Single Inst, Multi Data (SIMD)



یک دستور خوانده می‌شود و Data هایش چندتا اند

هرکدام از این واحدها می‌توانند مستقل با CPU کار کنند

کارهایی که خیلی پردازش روی Data داریم

برخلاف MIMD یک واحد کنترل دارند و همه یک دستور

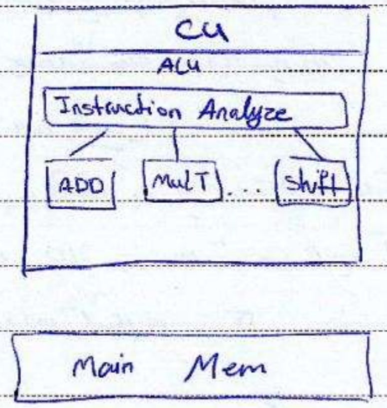
را انجام می‌دهند مثلا JPEG کردن عکس و کارهای

multi media

شیوه‌ی آدرس دهی در پردازنده‌های 386 ، 486 چگونه است
و تا چه مقدار از حافظه را می‌توانند آدرس دهی کنند

Subject:
Year. Month. Date. ( )

Multi Inst single Dat



چندین دستور را می‌خوانند (داده‌های کوتاه)

Data کم ، Inst زیاد و برای خواندن

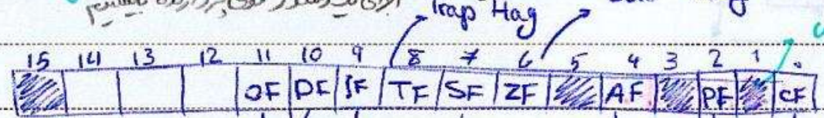
Data باید Inst های را انجام دهد

کاربرد در محاسبات پیچیده (ماتریس معکوس) که یک بار Data را بخوانند و پردازش‌های زیادی انجام دهد یک دستور از چندین دستور مجموعه‌های انجام می‌شود

یک Data یکبار fetch می‌شود

SISD (Single Inst , Single Data)

Flag Register (16 بیتی)



Trap Flag: 1 = هر وقت بخواند دستور اجرا کند یک دستور اجرا می‌شود و بعد از اجرای یک دستور توی پردازنده ببینیم

Zero Flag → (=1 , AX=0 , 0=2 AX<>0)

overflow Flag: در تا مثبت جمع شود منفی / منفی و ... مثبت

Interupt Enable: اگر یک باشد امکان Interupt وجود دارد

parity Flags: چون Data زیاد خوانده می‌شود

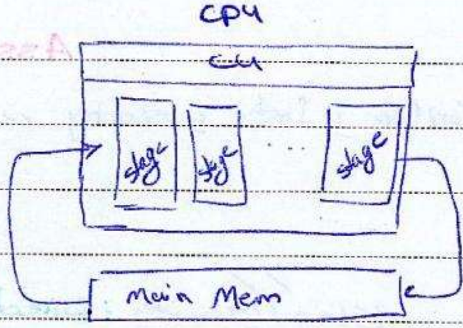
carry Flags: دو تا ۱۶ بیتی جمع شود AX وقتی carry ایجاد کند ۱ می‌شه

Direction: توی array حرکت می‌دهد و اعمال لازم را از اول به آخر یا بعدی بخود بتوان اضافه می‌کند یا کم (Direction) از آنجا این می‌باشد یا صفر یعنی به جلو می‌رود یا به عقب کم کند مقدار اندیس را
DI ~ Destination Inden
SI ~ Source "

Auxicary carry Flag: اگر از بیت ۷ به ۸ ، Carry ایجاد شود
این ۱ می‌شود یعنی نشان می‌دهد carry
بین ۷ و ۸ است

12, 13, 14 → ~~Stepie~~

I/O previlag lever
&
Nested Task



pipe line خط لوله

cpu از stageها تشکیل شده

stageها یک clock تفاوت دارند بین اینها

یک buffer قرار دارد که اگر نتیجه حاضر بود به بعدی بدهد

شبیه خط تولید کارخانه

Instruction را از حافظه بخوان بگذار در buffer سپس از buffer بخوان تفکیک کن و سپس نتیجه را بگذار در buffer سپس از buffer بگذار در حافظه

باعث افزایش سرعت پردازش می‌شود

هر واحد وابسته به واحد قبل است و همه واحدها در هر زمان فعال هستند ← فرض می‌کنیم هر stage، یک clock طول می‌کشد

بعد m clock، نتیجه اول است، clock بعدی نتیجه دوم و ... و بعد از $m+(n-1)$ clock نتیجه کل حاضر است

اگر می‌خواست در cpuهای قدیمی اجرا شود $m \times n$ clock طول می‌کشید

برای اجرای برنامه‌هایی که دستوراتش sequential است خوب است مثلاً اگر دستور jump باشد نمی‌شود از این استفاده کرد → conditional jumpها که هستند (براساس flag کار می‌کنند)

در کامپیوترهای قدیمی دستور اول خوانده می‌شود، اجرا می‌شود و نتیجه برمی‌گردد بعد دستور بعدی خوانده می‌شود

ولی در این نوع همه دستورات به موازات هم هستند و فقط منتظر result قسمت قبلی هستند

Subject:
Year. Month. Date. ( )

Assembly

Assembly { Instruction ها / Directives

هر دستور در assembly یا Inst. یا Dire است

Directive: دستورهایی که برای assembler استفاده می‌شود

Instruction: دستوراتی به زبان ماشین است

Instruction : [Lable:] Mnemonic [operand] [; comment]

(Instruction دستور خود)

operand ها می‌توانند ۰، ۱، ۲ تا باشند

Directive : [Name] Directive [operand] [; comment]

(Directive دستور خود)

LABEL1: MOV AX, BX ; This ints. moves BX to AX

برای jump کردن به Lable جامپ می‌کنیم

Naming {
- 31 character
- A → Z , a → z
- 0 → 9
- . , $, _ , @

Directive : Instruction های Assembler هستند و برای Assembler معنی دارند

در assembly ، case sensetive نیست — تبدیل به Inst می‌شوند compile / تا اندکی را تبدیل می‌کنند

ASSUME	INCLUDE	EQU
ENDP	ORG	=
PROC	SEGMENT	EXTERN
PAGE	EVEN	PUBLIC
TITLE	DB → Define Byte / 1byte	.286C
SUBTTL	const { DW → " word / 2byte	.8088
EN8DS	DD → " double / 4byte	.286P
	EQ	.8086
	END	

Subject:

Year. Month. Date. ( )

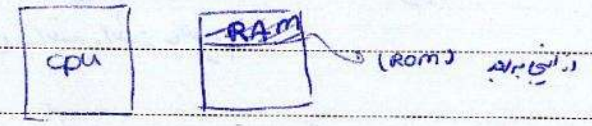
خروجی assemble ← binary ← LST که اگر error داشته باشد ـ نشان می‌دهد البته این زبان ماشین و اسمبلی است

ORG ← Directive

آدرس 16 بیتی می‌گیرد ، از اینجا به بعد اگر دستوری می‌نویسیم که بگوییم به این آدرس برود (مشخص کننده‌ی محل کد در حافظه) برنامه را در هر جای حافظه می‌توانیم load کنیم ـ همین ORG برای آدرس دهی کردن استفاده می‌کنیم

DB ← می‌تواند یک hex code که درست یعنی هم نقش Data ، هم نقش Inst

عدد اعشاری که دارید byte ها جا نمی‌شود



هر بیت از حافظه یک آدرس 16 بیتی دارد ← در کف اجرا می‌شود 24 بیتی

assemble → دستورات را ترجمه می‌کند
→ آدرس دستورات در حافظه (که این‌ها در کنار هم هستند)

{ binary b
decimal D
hex H }

اسم proc { procedure شروع
" ENP | خاتم

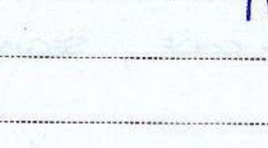
نگه‌داری اعداد مقایسه در حافظه 100

برای نگهداری بیش از 1 بایت باید یکی از این روش‌ها استفاده کرد

{ Little - endian ← برعکس ذخیره می‌کند
Big - endian } در حافظه اول bit بالا ذخیره می‌شود

که 8 بایت پایین بیشتر کاری کنیم پس کم‌تر است چون جدید باید اینکه کی کنیم

در مثال گذشته 1ABOH ذخیره کرد اول قسمت بزرگ بعد کوچک که برای کار به خوب است



PAPCO

assignment یک نوع بزرگ به کوچک

...tives

EVE... ← اگر خواستیم دستور ها بعدی خواستیم مقداری از حافظه را allocate کنیم از ا...

...شروع کنند.

PAGE ، TITLE ، SUBTTL : ← LST حالت پایینی بین compiler ...

حالت متنی دارد List file

در فایل LST هر page ۱۶ خط و هر خط ۸۰ کاراکتر باشد ← PAGE 16, 80

هرچی بنویسیم به ابتدای نتایج (کل فایل) کار برود ← TITLE ' '

بعد از TITLE را استفاده کردیم بعدش باید از sub استفاده کنیم

چندتا SubTTL ولی یک TITLE می توانیم داشته باشیم

SEGMENT ← در صورتی که ... code seg ... می شود که segها به ترتیب تعریف داخل حافظه ... شوند

SEGMENT Align_Type {byte, word, PARA, PAGE} combine_Type {public, STACK, COMMON} class Type

(از این آدرس شروع شود — از هر آدرس — از آدرس زوج — آدرس مضرب 16 — 256)

(STACK: مربوط به seg ... ; COMMON: می توانند از ناحیه مشترک استفاده کنند ... چندتا code seg ...)

سگمنت هایی هستند داخل کجا ... است با این مشخص می شود

...DE SEGMENT Byte

Type (COMMON): سگمنت های همنام از یک ناحیه از حافظه به صورت مشترک استفاده کنند

امکان overwrite وجود دارد — اگر این ... همنام باشند

معمولاً به صورت توضیحی است ولی باید نوشته شود — 'code' 'STACK' 'Data'

code از نوع seg

آخر seg با کلمه ENDS یا END SEGMENT مشخص می شود

کلمه seg صرفاً محدوده seg را مشخص کنند

...G SEGMENT Byte Public 'data'

DB 31

Subject:
Year. Month. Date. ( )

مشخص کردن seg

1. scope هر دو را باید بشکند که seg را مشخص کند (استفاده از دستور segment و ENDS)
2. از دستور Assume استفاده کنیم که مشخص کنیم نوع seg چی است، code ...
Directive

3. مقداردهی کردن
DS
CS
ES
SS
← instruction

کدام یک مربوط به کدام است، یعنی active است به کدام seg است
← دستوراتی مثل MOVE برای مقداردهی استفاده می‌کنیم

ASSUME

در کد می‌توان چندین بار از Assume استفاده کرد

یکی از register های seg

```
ASSUME | DS:
       | ES:
       | CS: CODESEG
       | SS:
```

CS: CODESEG → generate به codeseg آدرس‌ها بر اساس این می‌شود.

که کامپایلر ... برای code استفاده کند

```
ASSUME  DS: DATASEG
```

این دو تا دستور را می‌توانیم به یک خلاصه کنیم

```
ASSUME  DS: DATASEG, CS: CODESEG
```

معمولا ابتدای code است (ترتیب مهم است)

END ← در انتهای برنامه اصلی از END (assembly code) استفاده می‌کنیم

END [ ] ← Name مربوط به این دستور

= , EQU , DB

```
pi  EQU 31
pi  DB  31
pi  =   31
```

مساوی را چندین بار می‌توان استفاده کرد در صورتی که EQU فقط یکبار استفاده کرد، EQU هم می‌تواند هم string هم عدد را نسبت بدهد ولی = فقط برای عددهاست

P4PCO

Subject:
Year. Month. Date. ( )

اگر از مساوی استفاده کرده باشیم در جای جای برنامه می‌شود مقدار آن را تغییر داد ولی اگر از EQU
استفاده کنیم در کل برنامه دیگر نمی‌توان مقدار آن متغیر را تغییر داد

اگر با EQU یک متغیر مقدار دادیم دیگر نمی‌توانیم مقدار   ST EQU 'string'
آن متغیر را تعریف کنیم   ST = 'string'

تفاوت DD با DB: باعث می‌شود مقداری از حافظه را به خود اختصاص بدهد ولی = و EQU هیچ مقداری
از حافظه را به خود اختصاص نمی‌دهند

proc ← proc مثل seg است
که اگر یکی باشد می‌تواند اسم نداشته باشد

اسم   proc { NEAR → by default / FAR

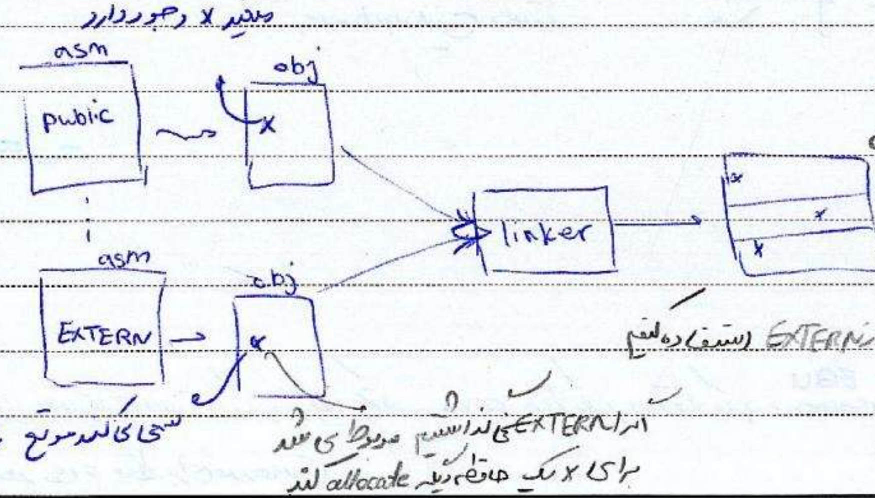
در آخر باید از ENDP استفاده کنیم

procedure برای اینکه همدیگر را call کنند اگر در یک seg باشند ← NEAR
در اصطلاحات ما چندین seg باشند ← FAR

اصلا منظور از محل call شدن procedure جایی که به تعریف شده یعنی اینکه در دو seg متفاوت و یا در یک seg
باشند

.286C ← assembler که داریم استفاده می‌کنیم مربوط به پردازنده 286 است و compiler خودش را
با این تنظیم می‌کند و باید در ابتدای کد باشد

متغیر x وجود دارد



PUBLIC و EXTERN ←

وقتی برنامه از چند فایل تشکیل شده
باشد اگر بخواهیم یک x که الان
استفاده می‌کنیم قبلاً تعریف
کردیم باید در یکی از فایل‌های دیگر از EXTERN استفاده کنیم

خطا می‌کند موقع compile پیدا نکند

اگر EXTERN نگذاشتیم مربوط می‌شد
برای x یک حافظه دیگر allocate کند
یعنی دو تا x با هم تفاوت داشتند

PAPCO

Subject:
Year. Month. Date. ( )

```
DATASEG SEGMENT Byte Public 'data'
    A DB 40
    B DB 14, ?, 0AH
    B DB 'This is a string'
    C DW ?
ENDS
```

A DB 40 → متغیری یک بایتی با مقدار 40

B DB 14, ?, 0AH (عنصر اول: 14) → آرایه سه عنصری، عنصر دوم مقدار دهی نشود

C DW ? → متغیر از نوع int، 2 بایتی که مقدار ندارد

B DB 20 DUP(0) = B DB 0,0,0,... (20)

DUP: duplicate

B DB 20 DUP(40, 4 DUP(' '), 40)

آرایه ای که در اول و آخرش 40 است و وسط آن 4 تا space که 20 بار پشت هم تکرار می شود

```
B DB 14, ?, 0AH
  DB 20,21
```

→ شکل اینکه DB را یکبار با کاما استفاده کنیم

HERE → به آدرس جاری، آدرسی که الان در آن قرار گرفتیم

Directive: عبارت خاص که فقط استفاده می شود

identifier (id)

```
HERE-LABLE DW HERE
MOV AX, 0
```

→ HERE-LABLE : MOV AX, 0

برای اینکه بتوانیم بعدا به این آدرس jump کنیم

```
CODESEG SEGMENT byte public 'code'
Assume CS: CODESEG, DS: DATASEG
MOV BX, CODESEG
MOV CS, BX
:
:
ENDS
```

CODESEG SEGMENT: یک code seg تعریف کنیم

Assume: مشخص می شود seg از نوع code است

MOV BX, CODESEG: برای اجرای برنامه باید code seg مشخص شود

MOV CS, BX: cs مقدار دهی شد که نشان دهنده code seg است

: کد برنامه

Subject:
Year. Month. Date. ( )

operations

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

داخل این می‌تواند متغیر table باشد

Directive

- SEG( ) و آدرس ابتدای seg که را برمی‌گرداند به دست می‌شود
- OFFSET ( )
- LENGTH( )
- TYPE ( ) → تایپ
  - 1 → byte
  - 2 → word
  - 4 → Double
  - −1 → NEAR
  - −2 → FAR

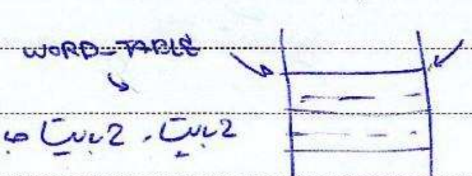
A DB SEG CS:BP → آدرس را در A می‌ریزد

B DB TYPE A → Type A در B نوشته می‌شود

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

FIRST_BYTE EQU PTR BYTE WORD_TABLE →

pointer



FIRST-BYTE → ابتدای word Table است و می‌گوید که
اگر از این طریق استفاده کنیم یک بایت یک بایت جابجا می‌شود

2 بایت، 2 بایت جابجا می‌شود

PAPCO

(array)