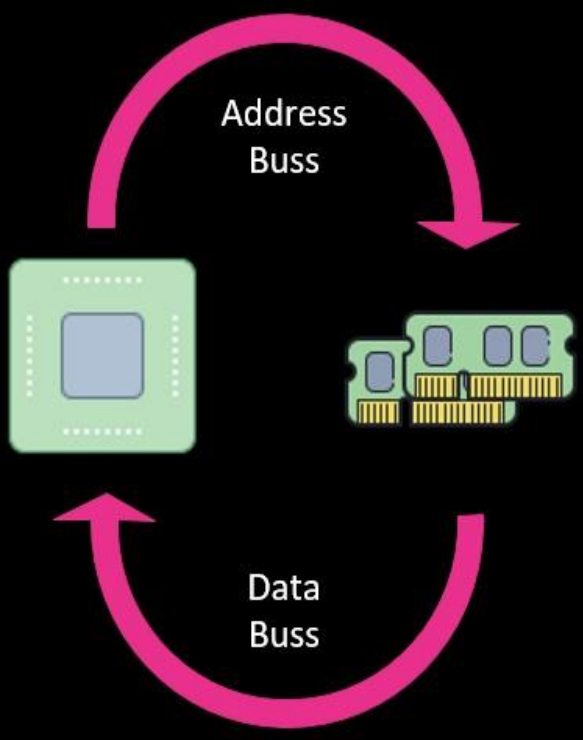
در سال 1978 شرکت Intel اولين ريزپردازنده 16 بيتي خود را تحت عنوان 8086 روانه بازار کرد.
تمام رجيستر هاي داخلي، گذرگاه داده و دستور العمل هاي 8086 شانزده بيتي بودند. البته گذرگاه آدرس اين ريزپردازنده 20 بيتي بود و به همين دليل تا يک MB حافظه را مي توانست آدرس دهي کند.



توضیح	نام رجیستر	نام کلی	بایت کم ارزش	بایت پر ارزش	
محاسبات و I/O	Accumulator	AX	AL	AH	گروه داده
نگه داری آدرس (آفست DS)	Base	BX	BL	BH	
شمارش دفعات تکرار	Count	CX	CL	CH	
آدرس پورت	Data	DX	DL	DH	
آفست SS	Stack Pointer	-	SP		گروه اشاره گر
	Base Pointer	-	BP		
عملیات های رشته ای برای DS (String commands)	Source index	-	SI		
	Dest. index	-	DI		
آفست CS	Instruction Ptr.	-	IP		
آدرس شروع بخش Extra در حافظه	Extra Segment	-	ES		گروه سگمنت
آدرس شروع بخش Code در حافظه	Code Segment	-	CS		
آدرس شروع بخش Data در حافظه	Data Segment	-	DS		
آدرس شروع بخش Stack در حافظه	Stack Segment	-	SS		
توضیحات در **صفحه 3**	Status & Control Flags	-	Flags L	Flags H	پرچم کنترل و وضعیت

تمام ثبات های 8086، 16 بیتی اند.

ثبات AX :

10110010	10110010
AH	**AL**

قرار دادن مقدار 2F در AH :

MOV AH,2F

هر کدام از ثبات های موجود در 8086 تک منظوره اند و کارایی خاص خود را دارند.

ثبات های گروه داده:

- ثبات AX در اعمال ورودی خروجی و محاسبات استفاده می شود.
- ثبات BX برای نگه داری آدرس است. در محاسبات نیز به ما کمک می کند.
- ثبات CX برای شمارش دفعات تکرار (مثلاً) یک حلقه است. در محاسبات نیز به ما کمک می کند.
- ثبات DX برای ذخیره سازی آدرس پورت است. در محاسبات نیز به ما کمک می کنند. (اعمال ضرب و تقسیم با اعداد بزرگ)

❖ از ثبات AX برای انجام محاسبات استفاده می شود ولی از ثبات های BX، CX و DX نیز می توان مانند AX برای انجام محاسبات استفاده کرد.

در پردازنده روبرو:

Bus Interface Unit دستورالعمل ها و داده هایی که نیاز داریم و قرار است به واحد EU بفرستیم را برای CPU فراهم می کند.

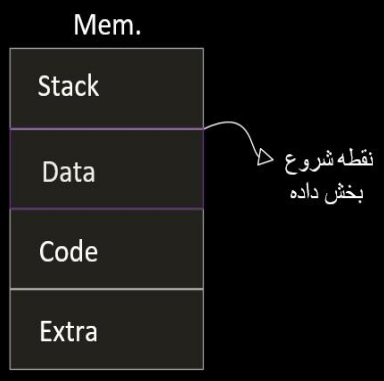
Execution Unit دتورات را اجرا می کند.

داده ها => numbers that mean things

دستورالعمل ها => numbers that do things

هر برنامه در زبان اسمبلی از 4 قسمت (سگمنت) تشکیل می شود:

- Stack Segment : برای داده های موقت است و اول از همه یک فضای رزرو شده برای این قسمت در حافظه مشخص می کنیم.
- Data Segment : داده های برنامه در این قسمت قرار می گیرند. به عنوان مثال متغیر ها.
- Code Segment : منطق برنامه.
- Extra Segment : داده های بیشتر.

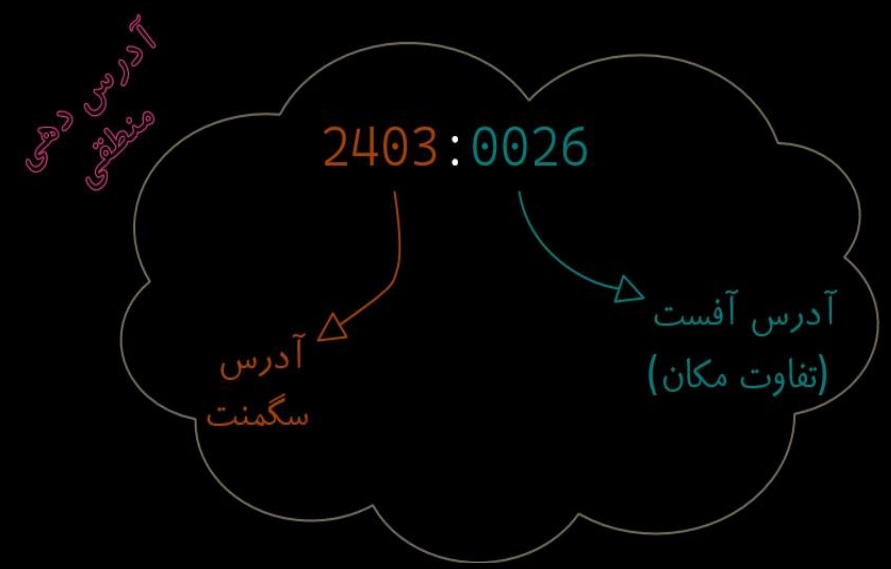


ثبات های گروه سگمنت:

- ثبات SS (Stack Segment) آدرس شروع بخش پشته برنامه (موجود در حافظه) را در خود نگه می دارد.
- ثبات DS (Data Segment) آدرس شروع بخش داده برنامه (موجود در حافظه) را در خود نگه می دارد.
- ثبات CS (Code Segment) آدرس شروع بخش کد برنامه (موجود در حافظه) را در خود نگه می دارد.
- ثبات ES (Extra Segment) آدرس شروع بخش اضافه برنامه (موجود در حافظه) را در خود نگه می دارد.

حال فرض کنید که آدرس شروع بخش داده یک برنامه موجود در حافظه، 2403 باشد.

برای مشخص کردن یک بخش خاص (مثلاً یک متغیر) موجود در بخش داده، آدرس آن را به صورت زیر می نویسیم:



آفست چیست؟ فرض کردیم که "بخش داده موجود در حافظه" از آدرس 2403 شروع شده است.

سوالی که مطرح می شود این است که حال متغیر (مثلاً X) موجود در "بخش داده" را چگونه مکان یابی کنیم؟

برای این کار فرض می کنیم که داخل "بخش داده" قرار داریم و و سر تا ته این بخش را می توانیم از 0000 (هگز) تا FFFF (هگز)، آدرس دهی کنیم. حال اگر متغیر مورد نظر در آدرس 0026 قرار داشته باشد، می توانیم بگوییم که آدرس منطقی متغیر مورد نظر، 2403:0026 است.

❖ در 8086، آدرس باس 20 بیت است. یعنی حافظه می تواند از 00000 (هگز) تا FFFFF (هگز) آدرس بگیرد.

حال سوالی که مطرح می شود این است که بخش سگمنت آدرس دهی منطقی 4 رقم هگز دارد ولی چرا حافظه از 00000 آدرس می گیرد؟
در آدرس دهی منطقی، صفر آخر بخش آدرس سگمت را نادیده می گیریم (پس 2403 در واقع 24030 است).

پس با این اوصاف برای تبدیل آدرس منطقی به آدرس فیزیکی از فرمول زیر استفاده می کنیم:

```
 24030
+ 0026
------
 24056
```

توجه داشته باشید که هر رقم هگز معادل 4 بیت است. همانطور که می توانید مشاهده کنید، آدرس فیزیکی متشکل از 5 رقم هگز یا همان 20 رقم باینری است. پس به کمک محاسبات زیر می توان نتیجه گرفت که با داشتن یک آدرس باس 20 بیتی، می توان یک حافظه یک MB ی را آدرس دهی کرد:

$20^2 \div 1024 = 1\ MB$

توجه: رجیستر CS نیز مانند بقیه رجیستر ها 16 بیتی است و (به عنوان مثال) عدد 2403 در آن قرار میگیرد نه 24030 عدد صفر، آخر 2403 وجود دارد ولی به دلیل جاگیر بودن، آن را نمی نویسیم.

سوال: آفست در کدام رجیستر قرار می گیرد؟ در این سناریو IP
پس آدرس منطقی 2403:0026 در رجیستر های CS:IP قرار می گیرد.

❖ پس کار رجیستر های اشاره گر، ذخیره سازی آدرس آفست است.

ثبات های گروه اشاره گر

- IP (Instruction Pointer) برای CS (Code Segment)
- SI (Source Index) و DI (Dest. Index) و BX (Base) موجود در گروه داده) برای DS (Data Seg.)
- SP (Stack Pointer) و BP (Base Pointer) برای SS (stack Seg.)

نکته1: ثبات IP کمی tricky است و گاهی اوقات حتی این ثبات را یک ثبات اشاره گر نمی دانند. IP همراه با CS به دستورالعمل بعدی (که باید توسط پردازنده اجرا شود) اشاره می کند.

نکته2: BX آفست را نگه می دارد و از SI و DI برای عملیات های رشته ای استفاده می شود.

نکته3: SP آفست را نگه می دارد و BP برای دسترسی به متغیر هایی است که در پشته قرار دارند.

جدول بیت های ثبات 16 بیتی پرچم

Flag H								Flag L							
X	NT	IO	PL	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	**CF**

موقعیت بیت	نام پرچم	عملکرد
0	CF (Carry)	پرچم نقلی: اگر بر بیت پر ارزش نتیجه[1] ، carry یا borrow اتفاق افتد، پرچم 1 شده و در غیر این صورت 0 خواهد بود
2	PF (Parity)	پرچم توازن: این پرچم 1 می شود اگر تعداد بیت های 1 در هشت بیت مرتبه پایین نتیجه، زوج باشد. در غیر این صورت 0 می شود.
4	AF (Auxiliary)	اگر بر چهار بیت کم ارزش AL، carry یا borrow اتفاق افتد، مقدار این پرچم برابر 1 و در غیر این صورت 0 می شود.
6	ZF (Zero)	پرچم صفر: اگر نتیجه صفر باشد، این پرچم 1 و در غیر این صورت 0 می شود.
7	SF (Sign)	پرچم علامت: این پرچم مقدار پر ارزش ترین بیت نتیجه را می گیرد. (بیت علامت)
8	TF (Trap -or- Trace)	پرچم تک گامی: وقتی این پرچم 1 باشد، بعد از اجرای دستورالعمل بعدی، یک single-step اتفاق می افتد. با به وجود آمدن وقفه single-step، این بیت 0 می شود. می توان 1 بودن این بیت را به فعال بود حالت debug mode تشبیه کرد. بدین معنا که به هنگام اجرا شدن یک دستورالعمل، سیستم صبر کرده و مستقیماً به دستورالعمل بعدی نمی رود.
9	IF (Interrupt)	پرچم فعال سازی وقفه: وقتی این پرچم 1 شود، وقفه های قابل پوشش باعث می شوند که CPU کنترل برنامه را به مکان بردار وقفه منتقل کند. (اگر وقفه داشته باشیم، این پرچم 1 می شود)
10	DF (Direction)	پرچم جهت: 1 بودن این پرچم، موجب می گردد که دستورات رشته ای (string) به طور خودکار ثبات نمایه مربوطه را کاهش دهند و اگر 0 باشد، افزایش خودکار صورت می گیرد. (اگر 1 باشد یعنی عمل انتقال یا مقایسه از راست به چپ و اگر 0 باشد یعنی عمل انتقال یا مقایسه از چپ به راست است)
11	OF (Overflow)	پرچم سرریز: اگر نتیجه محاسبات علامت دار انجام شده قابل قرار گرفتن در تعداد بیت های عملوند مقصد نباشد (سرریز رخ دهد)، این پرچم 1 می شود.

1.. نتیجه (به عنوان مثال) می تواند حاصل جمع دو عدد (بدون در نظر گرفتن carry) باشد.

عملوند = داده ها

عملگر = مثلاً عملیات جمع

مثال:

```
  1001 1100
+ 0110 0100
-----------
  0000 0000
```

CF=1 ZF=1
PF=1 SF=0
AF=1

روش های آدرس دهی در 8086

ثبات های معتبر	توضیح	مثال	حالت آدرس دهی
	عملوند مبدا[1] یک مقدار ثابت است (مقدار 100 را در ثبات AX ذخیره کن)	MOV AX,100	فوری Immediate
–	منبع و مقصد داده، ثبات های CPU است (محتویات BX را در AX ذخیره کن)	MOV AX,BX	ثباتی Register
	1000H آدرس آفستی است که به مقدار مورد نظر در data segment اشاره دارد (محتویات مکان 1000H را در ثبات AH ذخیره کن)	MOV AH,[1000H]	مستقیم Direct
[BX] [BP] [DI] [SI]	در این روش، آدرس (مثلا) 1000H در یکی از ثبات های base یا index (BP, BX, DI, SI) قرار دارد یاداوری: SI، DI و BX برای DS و BP برای SS بود پس اگر 1000H در BP قرار داشته باشد، آدرس آفستی خواهد بود که به یک مقدار در stack segment اشاره دارد و اگر 1000H در SI قرار داشته باشد، آدرس ایندکسی خواهد بود که به یک مقدار در data segment اشاره دارد (1000H در BX قرار دارد پس محتویات مکان DS:1000H را در CL ذخیره کن)	MOV CL,[BX] MOV CL,(BX)	غیرمستقیم ثباتی Register indirect
d+[BX], d+[BP], d+[DI], d+[SI]	همان قبلی است با این تفاوت که یک displacement نیز وجود دارد (1000H در BX قرار دارد پس محتویات مکان DS:1014H را در AX ذخیره کن)	MOV AX,[BX]+14 MOV AX,[BX+14]	نسبی (پایه) Based or Indexed

[BX][DI]
[BX][SI]
[BP][DI]
[BP][SI]
	در این روش، آدرس آفست موجود در یک ثبات base (BX, BP) را با آدرس ایندکس موجود در یک ثبات index (DI ,SI) جمع می کنیم
	(سومی: یک آدرس در BP و یک آدرس در DI قرار دارد
	این دو آدرس را با یکدیگر جمع کن و محتویات آدرس حاصله که به یک مقدار در SS اشاره دارد را در AX ذخیره کن)
		MOV AX,[BX+DI]
		MOV AX,[BX+SI]
		MOV AX,[BP+DI]
		MOV AX,[BP+SI]
			نسبی (پایه)
			اندیس (شاخص) دار
			Base plus indexed

d+[BX][DI] d+[BX][SI] d+[BP][DI] d+[BP][SI]	همان قبلی است با این تفاوت که یک displacement نیز وجود دارد (یک آدرس در BP و یک آدرس در DI قرار دارد این دو آدرس را با یکدیگر جمع کن و آدرس حاصله را نیز با 13 جمع کن و در نهایت محتویات آدرس نهایی که به یک مقدار در SS اشاره دارد را در AX ذخیره کن) SS0+BP+DI+13	MOV AX,[BP,DI+13]	نسبی (پایه) اندیس (شاخص) دار به همراه جابجایی Base plus index with displacement

1.. عملوند مبدا،عملوند مقصد عملگر

MOV AX,100

ASSEMBLY

فرض کنید که می خواهیم یک متغیر به نام A تعریف کردن و چند بیت در آن ذخیره کنیم.

برای این کار از عبارت زیر استفاده می کنیم:

A DB 10101010b

DB چیست؟ DB یک رهنمون است و مشخص می کند عددی که می خواهیم

در متغیر A ذخیره کنیم، حد اکثر چند بیت می تواند باشد؟ (در این مورد 8 بیت یا 1 بایت)

رهنمون	حجم (بایت)
DB	1
DW	2
DD	4
DQ	8
DT	10

نکته: هر رقم در مبنا های HEX، DEC و OCT

به اندازه 4 بیت فضا اشغال می کند.

سناریو: تبدیل کد ASCII به BCD:

به عنوان مثال برای تبدیل کردن 31 (هگز) یا همان 0011 0001 (باینری)،

عدد مذکور را در 0F (هگز) یا همان 0000 1111 (باینری) ضرب می کنیم

که کد اسمبلی آن به صورت زیر است:

Char	ASCII (HEX)	ASCII (BIN)	BCD (HEX)	BCD (BIN)
0	30	0011 0000	00	0000 0000
1	31	0011 0001	01	0000 0001
9	39	0011 1001	09	0000 1001

توجه: string، رشته یا همان کاراکتر 9 در جدول ASCII معادل 39h است و

عدد 9 در جدول ASCII معادل 09h است.

```
A  DB  '2EEE'
B  DB  4  DUP(?)

MOV  BX,OFFSET A
MOV  DI,OFFSET B

MOV  CX,4h
L:
MOV  AX,[BX]
AND  AX,0F0Fh
MOV  [DI],AX
ADD  BX,2h
ADD  DI,2h
LOOP  L
```

تحلیل کد:

رهنمون DB مشخص می کند که اندازه پنجره متغیر A، یک بایت است.

معنای عبارت A DB 2Bh این است که عدد 2Bh عیناً داخل متغیر A قرار می گیرد.

معنای عبارت A DB '2B' این است که string (رشته) 2B در قدم اول به معادل ASCII (HEX) آن تبدیل شده و سپس توسط متغیر A نمایش داده می شود.

پس در قدم اول '2B' به 32 42 (هگز) تبدیل شده و سپس 32h توسط متغیر A نمایش داده می شود.

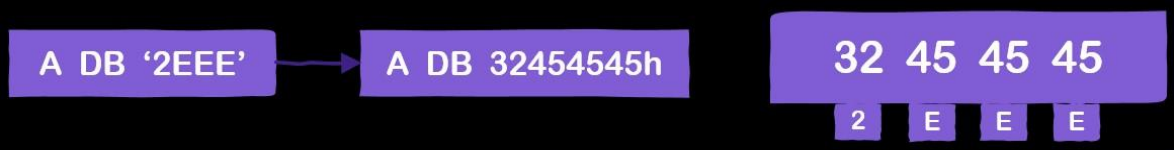
(از این جهت که رهنمون یک بایتی DB را مورد استفاده قرار داده ایم، فقط 32h توسط متغیر A نمایش داده می شود)

به عنوان مثال اگر از رهنمون دو بایتی DW استفاده می کردیم، 3242h به شکل توسط متغیر A نمایش داده می شد.

نکته مهم: این مسئله فقط در مورد رشته ها صادق است. بدین معنا که اگر متغیر A از رهنمون DB استفاده کند و بخواهیم عدد 101010101b که 9 بیت است را در A ذخیره کنیم، با ارور مواجه می شویم. ولی اگر بخواهیم رشته 2B که 16 بیت است را در A ذخیره کنیم، مشکلی وجود نخواهد داشت زیرا A مانند ماشین تورینگ عمل خواهد کرد.

در کل بهتر است که به هنگام ذخیره کردن رشته ها در یک متغیر، خیلی به مغز خود فشار نیاوریم و از رهنمون DB استفاده کنیم.

ولی به هنگام ذخیره کردن اعداد باید مطمئن شویم که اندازه عدد از اندازه رهنمون بزرگ تر نیست.



توجه: هر دو رقم هگز، یک بایت فضا اشغال می کند.
هر یک کاراکتر رشته ای، یک بایت فضا اشغال می کند.

حال در خط دوم برنامه، متغیر B را تعریف می کنیم و از این جهت که فعلاً می خواهیم این متغیر خالی باشد، چهار تا null یا همان ? یا همان 00h در آن ذخیره می کنیم.

(چهار، طول رشته موجود در متغیر A است و علت این که می خواهیم طول B نیز چهار باشد این است که محتویات A بعد از اعمال محاسبات روی آن، در B ذخیره می شود)

آیا می توانستیم از عبارت B DB 00h و یا B DB ???? نیز استفاده کنیم؟ خیر زیرا در این حالت نمی توان به یک متغیر، چهار تا 00 یا ? خوراند.

5 DUP(?) یعنی 5 تا 00h در متغیر B قرار بده.

آدرس متغیر A را در BX ذخیره می کنیم.

آدرس متغیر B را در DI ذخیره می کنیم.

حال عدد 4 (هگز) را داخل رجیستر CX قرار می دهیم (04h در CL ذخیره می شود). اگر به خاطر داشته باشید، رجیستر CX برای شمارش دفعات تکرار (مثلاً) یک حلقه بود.

این کار باعث می شود که حلقع L، 4 بار تکرار شود.

آدرس متغیر A در BX قرار داشت. حال به کمک دستور MOV AX,[BX] محتویات این متغیر را در AX ذخیره می کنیم.

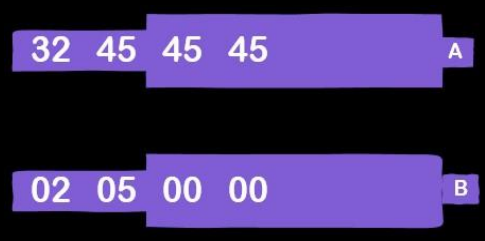
توجه: در زبان اسمبلی، داده های رشته ای، دوتا دوتا خوانده می شوند زیرا رجیستر AX یک رجیستر 16 بیتی است و هر رشته (مثلاً '2') در زبان اسمبلی، معادل 8 عدد صفر و یک (با توجه به جدول ASCII) است. پس دستور بالا باعث می شود که رشته '2E' یا همان 3245h در رجیستر AX ذخیره شود.

حال محتویات AX را در 0F0F ضرب می کنیم. پس 3245h به 0205h تبدیل می شود.

آدرس متغیر B در DI قرار داشت. حال به کمک دستور MOV [DI],AX محتویات AX را در متغیر B ذخیره می کنیم.

آدرس متغیر A در BX قرار داشت. حال به کمک دستور ADD BX,2h آدرس شروع متغیر A را 2 واحد به جلو سوق می دهیم.

آدرس متغیر B در DI قرار داشت. حال به کمک دستور ADD DI,2h آدرس شروع متغیر B را 2 واحد به جلو سوق می دهیم.



اگر به خاطر داشته باشید، هر برنامه در زبان اسمبلی از سه قسمت اصلی Stack، Data و Code تشکیل می شد.

دستورات SEGMENT و ENDS برای تعریف کردن یک segment در برنامه اند.

در این مثال، سه segment به نام های S (برای stack)، D (برای Data) و C (برای Code) تعریف کردیم.

توجه1: حجم Stack Segment را در این سناریو 32 بیت در نظر گرفتیم.

توجه2: دستور END موجود در انتهای کد، نشان دهنده پایان برنامه است.

توجه3: هر برنامه می تواند از چند procedure یا رویه تشکیل شود که در هر رویه تعدادی دستور قرار دارد و مجموعه این دستورات، یک کار خاص را انجام می دهد. مثل جمع کردن محتویات سه متغیر N1، N2 و N3 و قرار دادن حاصل محاسبات در متغیر SUM.

دستورات PROC و ENDP برای تعریف کردن یک رویه در برنامه اند. در این سناریو یک رویه بیشتر (به نام MAIN) نداریم.

توجه4: یک رویه می تواند دو حالت FAR یا NEAR داشته باشد. FAR مشخص می کند که می توانیم از segment های دیگر نیز داده برداریم.

توجه5: برای معرفی کردن هر یک از segment های تعریف شده (S، D و C) به ثبات های مربوطه، از دستور ASSUME استفاده می کنیم.

توجه6: از این جهت که نمی توان ثبات های Flag و DS را به شکل مستقیم مقدار دهی کرد، از ثبات AX برای قرار دادن آدرس متغیر D در ثبات DS کمک می گیریم.

```
S SEGMENT
DB 32 DUP(?)
S ENDS

D SEGMENT
N1 DB 1h
N2 DB 2h
N3 DB 3h
SUM DB ?
D ENDS

C SEGMENT
MAIN PROC FAR
ASSUME CS:C, DS:D, SS:S
MOV AX,D
MOV DS,AX
MOV AL,N1
ADD AL,N2
ADD AL,N3
MOV SUM,AL
MAIN ENDP
C ENDS
END MAIN
```

برای کوتاه کردن کد بالا، می توان از مدل small استفاده کرد:

```
.model small

.stack 32

.data
N1 DB 1h
N2 DB 2h
N3 DB 3h
SUM DB ?

.code
MAIN:
MOV AX,@data
MOV DS,AX
MOV AL,N1
ADD AL,N2
ADD AL,N3
MOV SUM,AL
END MAIN
```

کد زیر را تحلیل کنید:

```
.model small

.stack 32

.data
A  DB  5h,18h,12h
MAX  DB  ?

.code
MAIN:
MOV  AX,@data
MOV  DS,AX
MOV CX,3
MOV BX,OFFSET A
SUB AL,AL
AGAIN:
CMP AL,[BX]
JA NEXT
MOV AL,[BX]
NEXT:
INC BX
LOOP AGAIN
MOV MAX,AL
END  MAIN
```

متغیر A دارای سه مقدار 5h، 18h و 12h است و تصمیم داریم این سه مقدار را با یکدیگر مقایسه کرده و بزرگ ترین آن ها را در متغیر MAX بریزیم.

یاداوری: رجیستر CX برای شمارش دفعات تکرار یک حلقه بود و مقدار آن در این مورد 3 است.

آفست (آدرس شروع) متغیر A را در رجیستر BX ذخیره می کنیم.

محتویات رجیستر AL را صفر (00h) می کنیم.

اولین مقدار موجود در متغیر A را با محتویات رجیستر AL که در حال حاضر 00h است مقایسه می کنیم.

JA به معنای Jump if Above یا ">" است. پس از این جهت که 00h>05h نیست پس عمل jump صورت نمی پذیرد و مستقیم به NEXT نمی رویم. پس مقدار 05h در AL ذخیره شده و سپس به مرحله بعد که NEXT باشد می رویم.

محتویات BX، آفست (آدرس شروع) متغیر A بود و در این قسمت مقدرا BX را یک واحد افزایش می دهیم تا به مقدرا بعدی موجود در متغیر A (18h) اشاره کند.

حال به کمک دستور LOOP به AGAIN باز می گردیم (عمل LOOP دو بار دیگر رخ خواهد داد).

عملگر	عملوند مقصد	عملوند مبدا
MOV	AX	100

Jump instructions that test single flag

Instruction	Description	Condition	Opposite Instruction
JZ , JE	Jump if Zero (Equal).	ZF = 1	JNZ, JNE
JC , JB, JNAE	Jump if Carry (Below, Not Above Equal).	CF = 1	JNC, JNB, JAE
JS	Jump if Sign.	SF = 1	JNS
JO	Jump if Overflow.	OF = 1	JNO
JPE, JP	Jump if Parity Even.	PF = 1	JPO
JNZ , JNE	Jump if Not Zero (Not Equal).	ZF = 0	JZ, JE
JNC , JNB, JAE	Jump if Not Carry (Not Below, Above Equal).	CF = 0	JC, JB, JNAE
JNS	Jump if Not Sign.	SF = 0	JS
JNO	Jump if Not Overflow.	OF = 0	JO
JPO, JNP	Jump if Parity Odd (No Parity).	PF = 0	JPE, JP

Jump instructions for signed numbers

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JG , JNLE	Jump if Greater (>). Jump if Not Less or Equal (not <=).	ZF = 0 and SF = OF	JNG, JLE
JL , JNGE	Jump if Less (<). Jump if Not Greater or Equal (not >=).	SF <> OF	JNL, JGE
JGE , JNL	Jump if Greater or Equal (>=). Jump if Not Less (not <).	SF = OF	JNGE, JL
JLE , JNG	Jump if Less or Equal (<=). Jump if Not Greater (not >).	ZF = 1 or SF <> OF	JNLE, JG

Jump instructions for unsigned numbers

Instruction	Description	Condition	Opposite Instruction
JE , JZ	Jump if Equal (=). Jump if Zero.	ZF = 1	JNE, JNZ
JNE , JNZ	Jump if Not Equal (<>). Jump if Not Zero.	ZF = 0	JE, JZ
JA , JNBE	Jump if Above (>). Jump if Not Below or Equal (not <=).	CF = 0 and ZF = 0	JNA, JBE
JB , JNAE, JC	Jump if Below (<). Jump if Not Above or Equal (not >=). Jump if Carry.	CF = 1	JNB, JAE, JNC
JAE , JNB, JNC	Jump if Above or Equal (>=). Jump if Not Below (not <). Jump if Not Carry.	CF = 0	JNAE, JB
JBE , JNA	Jump if Below or Equal (<=). Jump if Not Above (not >).	CF = 1 or ZF = 1	JNBE, JA

<> - sign means not equal.

شیفت:

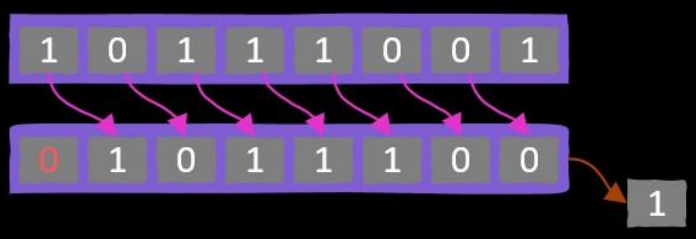
SHR1: محتویات بایت مورد نظر را یک واحد به سمت راست شیفت بده

و اول بایت را با صفر پر کن.

SHL1: محتویات بایت مورد نظر را یک واحد به سمت چپ شیفت بده.

و آخر بایت را با صفر پر کن.

SHR1:



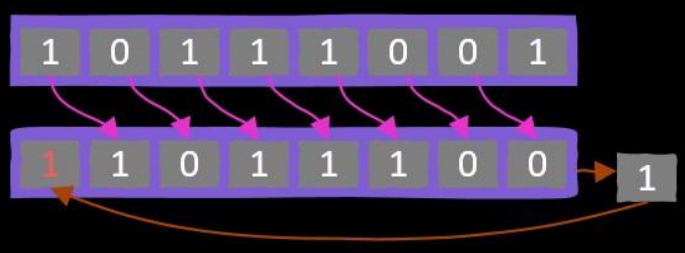
ROR1: محتویات بایت مورد نظر را یک واحد به سمت راست شیفت بده

و بیت بیرون افتاده را در اول بایت قرار بده.

ROL1: محتویات بایت مورد نظر را یک واحد به سمت چپ شیفت بده

و بیت بیرون افتاده را در آخر بایت قرار بده.

ROR1:



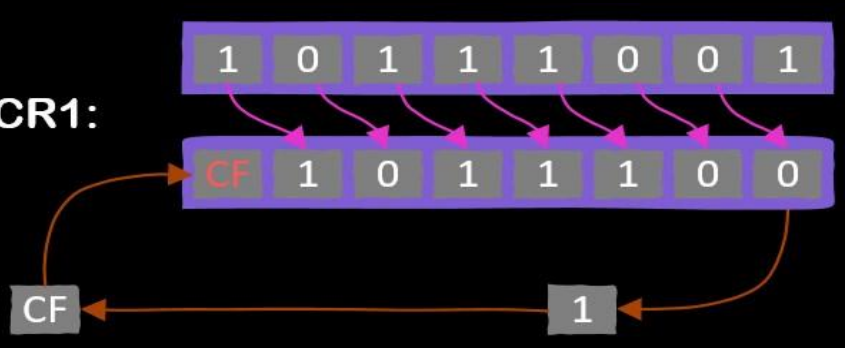
RCR1: محتویات بایت مورد نظر را یک واحد به سمت راست شیفت بده

و محتویات CF را در اول بایت قرار بده. سپس بیت بیرون افتاده را در CF ذخیره کن.

RCL1: محتویات بایت مورد نظر را یک واحد به سمت چپ شیفت بده

و محتویات CF را در آخر بایت قرار بده. سپس بیت بیرون افتاده را در CF ذخیره کن.

RCR1:



https://jbwyatt.com/253/emu/asm_tutorial_01.html

نیما احمدوند