بسم الله الرحمن الرحیم

آموزش زبان ماشین
و زبان اسمبلی

مولف :
مهندس مصطفی عبدالهیان

{کلیه مندرجات روی جلد}

{مخصوص شناسنامه کتاب}

{مخصوص سپاسگزاری}

ت

{مخصوص تقدیم}

فهرست مطالب

فصل اول سیستم اعداد و روش‌های کدگذاری

۱-۱ سیستم اعداد

در سیستم‌های عددی معمولی، موقعیت مکانی هر رقم دارای ارزش معینی است. در چنین سیستم‌هایی می‌توان عدد را به‌صورت زیر نمایش داد:

$$(a_{n-1}\ a_{n-2}\dots a_1\ a_0\ /\ a_{-1}\,a_{-2}\dots a_{-m})_b$$

در نمایش فوق داریم:

b: مبنا (Base) سیستم است و $0 \le a_K \le B-1$

n: تعداد ارقام صحیح

m: تعداد ارقام اعشاری

a_0، a_1، a_2 و... : ضرایب

هر عدد N در مبنای B، به‌صورت $(N)_B$ نمایش داده می‌شود. اگر مبنای عددی مشخص نگردد، ۱۰ منظور می‌شود.

مبناهای پرکاربرد:

دودویی	Binary	1 و 0
هشت هشتی	Octal	0 تا 7
دهدهی	Decimal	0 تا 9
شانزده شانزده‌ای	HexaDecimal	0 تا F

۱-۲ تبدیل مبناها

به علت آنکه در سیستم‌های کامپیوتری، مبناهای 2 و 16 پرکاربرد هستند، لازم است چگونگی تبدیل مبناهای مذکور به یکدیگر را مطالعه کنیم.

۱-۲-۱ نحوه تبدیل عدد از مبنای 10 به b

۱-۲-۱-۱ تبدیل از مبنای 10 به 2

- برای تبدیل اعداد صحیح دهدهی به دودویی از روش تقسیم متوالی استفاده می‌شود. در این روش عدد دهدهی بر 2 تقسیم می‌شود و باقیمانده و خارج‌قسمت محاسبه می‌گردند. اگر خارج‌قسمت صفر نباشد، خارج‌قسمت بر 2 تقسیم خواهد شد و این روند تا صفر شدن خارج‌قسمت ادامه می‌یابد. باقیمانده‌های ایجادشده از هر تقسیم، نگهداری می‌شوند و از آخرین باقیمانده به اولین باقیمانده در کنار هم نوشته

می‌شوند. عدد حاصل در مبنای 2 خواهد بود. بدیهی است که تقسیم به‌صورت صحیح انجام می‌شود. (خارج‌قسمت اعشاری نیست).

- برای تبدیل اعداد اعشاری از مبنای 10 به 2، باید قسمت صحیح و اعشاری را جداگانه به مبنای 2 تبدیل کنیم. برای تبدیل قسمت اعشاری، از روش ضرب متوالی در 2 استفاده شود. در این روش قسمت اعشار در 2 ضرب شده، قسمت صحیحِ حاصل، نگهداری می‌شود و این روند برای قسمت اعشاری حاصل ادامه می‌یابد تا قسمت اعشار به صفر برسد. سپس قسمت‌های صحیح حاصل را در کنار هم می‌نویسیم. عدد حاصل، تبدیل مبنای 2 قسمت اعشاری است. با تلفیق قسمت اعشاری و قسمت صحیح، عدد به‌طور کامل به مبنای دو تبدیل می‌شود.

روش ۱) تقسیم متوالی برای قسمت صحیح عدد

مثال الف) قسمت صحیح

$(23)_{10} = (?)_2$

23	2				
22	11	2			
❶	10	5	2		
	❶	4	2	2	
		❶	2	1	2
			❷	0	❷
				❶	

$= (10111)_2$

مثال ب) قسمت صحیح و اعشار

$(213/25)_{10} = (?)_2$

213	2						
212	106	2					
❶	106	53	2				
	❷	52	26	2			
		❶	26	13	2		
			❷	12	6	2	
				❶	6	3	2
					❷	2	❶
						❶	

ضرب متوالی برای قسمت اعشار عدد:

$0.25 \times 2 = ⓪.5$

$0.5 \times 2 = ①.0$

$= (11010101\ /\ 01)_2$

روش ۲) تبدیل مبنای 10 به 2 با روش تجزیه

برای این کار توان‌های 2 را از عدد بیرون می‌کشیم. از بزرگ‌ترین توان ممکن شروع کرده و هرگاه بتوانیم توان 2 را بیرون بکشیم در مکان متناظر با آن توان "1" قرار می‌دهیم. و در غیر این صورت "0" قرار می‌دهیم. این عمل تکرار می‌کنیم تا حاصل جمع توان 2 ها همان عدد موردنظرمان شود.

در مثال الف رنگ خاکستری در جمع عدد محاسبه نمی‌شود.

مثال الف)

$$213 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$$
$$= (1\ 1\ 0\ 1\ 0\ 1\ 0\ 1)_2$$

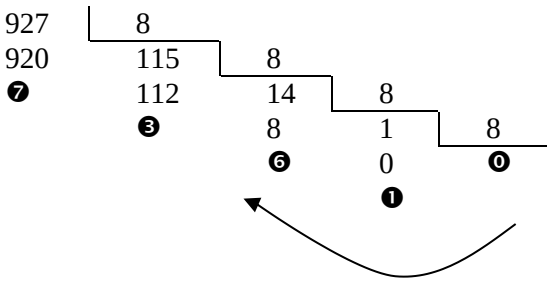
مثال ب)

$$43 = 32+8+2+1$$
$$= (101011)_2$$

۱-۲-۱-۲ تبدیل از مبنای 10 به 8

می‌بایست عدد در مبنای 10 را مرتباً تقسیم‌بر 8 کنیم و باقیمانده‌ها را نگه‌داریم. این عمل را آن‌قدر ادامه می‌دهیم تا خارج‌قسمت صفر شود. در آخر باقیمانده‌ها را از راست به چپ کنار هم می‌نویسیم.

$(954)_{10} = (?)_H$



۱-۲-۱-۳ تبدیل از مبنای 10 به H

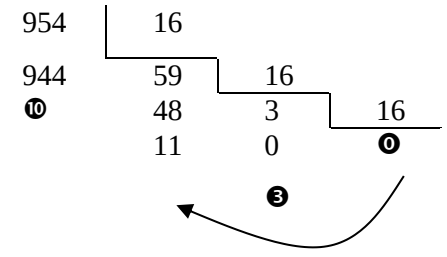
برای تبدیل اعداد ده‌دهی به هگزادسیمال کافی است عدد ده‌دهی را بر 16 تقسیم کنیم که باقیمانده اولین یا کم‌ارزش‌ترین، رقم هگزادسیمال است، سپس خارج‌قسمت را بر 16 تقسیم کنیم و باقیمانده را در نظر بگیریم، آن‌قدر این کار را ادامه دهیم تا خارج‌قسمت صفر شود.

بدیهی است اگر باقیمانده‌ها بزرگ‌تر از 9 باشند، رقم‌های هگزادسیمال A، B، C، D، E و F استفاده می‌شوند.

مثال الف) قسمت صحیح

$(954)_{10} = (?)_H$

$= (3BA)_H$

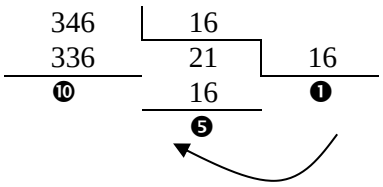


مثال ب) قسمت صحیح و اعشار

$(346/\ 26)_{10} = (?)_H$

$0/26 \times 16 = 4/16$
$0/16 \times 16 = 2/56$
$0/56 \times 16 = 8/96$

$= (15A/428)_H$



۱-۲-۲ تبدیل عدد از مبنای B به 10

روش کار : استفاده از نمایش ارزش مکانی

در این روش به هر قم یک موقعیت می‌دهیم. موقعیت‌ها برای قسمت صحیح از سمت راست به چپ و از صفر شماره‌گذاری می‌شوند و برای قسمت اعشار از سمت چپ به راست از 1- شماره‌گذاری می‌شوند. هر رقم را ضرب‌در مبنا (B) به توان موقعیت می‌کنیم و نهایتاً اعداد حاصل را باهم جمع می‌کنیم.

مثال) تبدیل مبنای 2 به 10

2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	
↑	↑	↑	↑	↑	↑		↑	↑	↑	
(1	0	1	1	0	1	/	1	1	1	$)_2 = (?)_{10}$

$$(1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$
$$+$$
$$\underset{0.5}{(1 \times 2^{-1})} + \underset{0.25}{(1 \times 2^{-2})} + \underset{0.125}{(1 \times 2^{-3})} = (45/875)_{10}$$

مثال) تبدیل مبنای 8 به 10

$(6402/\,54)_8 = (?)_{10}$

$$\underset{3072}{(6 \times 8^3)} + \underset{256}{(4 \times 8^2)} + \underset{0}{(0 \times 8^1)} + \underset{2}{(2 \times 8^0)} + \underset{0/625}{(5 \times 8^{-1})} + \underset{0/0625}{(4 \times 8^{-2})} = (3330/6875)_{10}$$

مثال) تبدیل مبنای 16 به 10

$(3BA)_{16} = (?)_{10}$

$$\underset{768}{(3 \times 16^2)} + \underset{176}{(11 \times 16^1)} + \underset{10}{(10 \times 16^0)} = (954)_{10}$$

۱-۲-۳ تبدیل عدد از مبنای 2 به 8، H و برعکس

برای تبدیل مبنای 2 به 8 کافی است (در قسمت صحیح عدد) رقم‌های عدد را از سمت راست، سه رقم سه رقم جدا کرد و به‌جای هر سه رقم مبنای دو، یک رقم مبنای هشت (۳ بیت) قرارداد. چنانچه تعداد رقم مضربی از سه نباشد باید به تعداد لازم در سمت چپ عدد صفر اضافه کرد که در مثال زیر این صفرها بارنگ قرمز مشخص‌شده‌اند.

در قسمت اعشار، به همان طریق انجام می‌دهیم فقط با این تفاوت که رقم‌های آن را از سمت چپ به راست جدا می‌کنیم.

رقم	ارزش ستون‌ها 4	2	1
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

$00(1\ 101\ 101\ /101\ 10)_2 0 = (?)_8 = (155/54)_8$

1 5 5 5 4

تبدیل مبنای 8 به 2 :

$(670/53)_8 = (?)_2 = (110\ 111\ 000/101\ 011)_2$

برای تبدیل مبنای 2 به H کافی است به‌جای هر رقم معادل 4 بیتی آن را از جدول زیر قرار دهیم:

رقم	ارزش ستون‌ها 8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
A (10)	1	0	1	0
B (11)	1	0	1	1
C (12)	1	1	0	0
D (13)	1	1	0	1
E (14)	1	1	1	0
F (15)	1	1	1	1

$00(10\ 1110\ 1101\ /1011\ 10)_2 00 = (2ED/B8)_H$

2 E D B 8

تبدیل مبنای 16 به 2 :

$(A0BC/D2)_H = (1010\ 0000\ 1011\ 1100\ /\ 1101\ 0010)_2$

۳-۱ اطلاعات ورودی به کامپیوتر

اطلاعاتی که وارد کامپیوتر می‌شوند ترکیبی از حروف A تا Z، ارقام ۰ تا ۹ و علائمی مثل $، #، *، + و ... است. این اطلاعات به روش خاصی در حافظه نگهداری می‌شوند. بعضی از این اطلاعات مقادیر عدد هستند، مثل شماره دانشجویی یک دانشجو و نمرات دانش آموزان. بعضی از این اطلاعات مقادیر عددی هستند، مثل شماره دانشجویی یک دانشجو و نمرات دانش آموزان. بعضی دیگر از این اطلاعات به‌صورت کاراکتری (رشته‌ای، حرفی) هستند، مثل نام دانشجو.

- نحوه نگهداری اعداد:

۱- عددی:

- صحیح
- اعشار

۲- رشته‌ای

- نحوه نگهداری اعداد صحیح در کامپیوتر:

۱- مقدار علامت (Sign & Value)

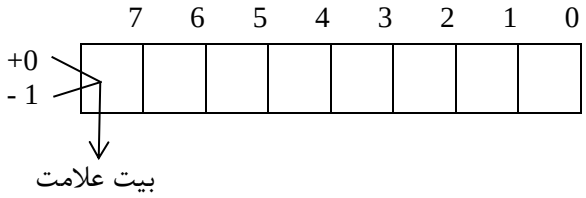
۲- مکمل (۱) (1's Complement)

۳- مکمل (۲) (2's Complement)

۱-۳-۱ نحوه نگهداری اعداد صحیح

۱-۱-۳-۱ روش مقدار – علامت (بیت علامت)

در این روش اعداد منفی به‌صورت اعداد مثبت نمایش داده می‌شوند و بیت سمت چپ عدد، علامت آن را نشان می‌دهد که اگر صفر باشد عدد مثبت است و اگر یک باشد عدد منفی است.



مثال) عدد 98- را به روش مقدار علامت در کامپیوتر با طول کلمات 8 بیت نمایش دهید.

$98 = 64 + 32 + 2$

1 1100010

نکته ۱) رنج اعداد قابل ذخیره در این روش در کامپیوتری با طول کلمات M بیتی برابر است با:

$-(2^{M-1}-1)$ تا $+(2^{M-1}-1)$

$-(2^{8-1}-1)$ تا $+(2^{8-1}-1)$ $\leftarrow$ $M=8$

-127 تا +128

نکته ۲) استفاده از این روش یک عیب بزرگ دارد و آن این است که برای عدد صفر دو نمایش وجود دارد.

0 000 0000
+ 0

1 000 0000
- 0

۲-۱-۳-۱ روش مکمل (۱)

در این روش برای نمایش عدد منفی کافی است عدد را به مبنای 2 برده و از آن مکمل(۱) بگیریم یعنی صفرها را به یک و یک‌ها را به صفر تبدیل کنیم. در این روش نیز بیت سمت چپ علامت عدد علامت عدد است.

مثال) عدد 87- را به روش مکمل(۱) در کامپیوتری با طول کلمات 8 بیتی نمایش دهید.

87 = 64 + 16 + 4 + 2 + 1

0 1 0 1 0 1 1 1 = (0 1 0 1 0 1 1 1)

مکمل(۱)

1 0 1 0 1 0 0 0

نکته: چون بیت علامت ندارد 8 بیت در نظر گرفته می‌شود.

- **نکته:** این روش نیز مانند روش قبل یک عیب عمده دارد و آن این است که برای صفر دو نمایش داریم:

1 1 1 1 1 1 1 1
+0

0 0 0 0 0 0 0 0
-0

۱-۳-۱-۳ روش مکمل(۲)

برای به دست آوردن مکمل(۲) می‌بایست ابتدا مکمل(۱) را به دست آورده سپس یک واحد به آن اضافه کنیم.

مکمل (۲) = مکمل (۱) + ۱

مثال) عدد 91- را به روش مکمل(۲) در کامپیوتری با طول کلمات 8 بیتی نمایش دهید.

91 = 64 + 16 + 8 + 2 + 1

0 1 0 1 1 0 1 1

مکمل(۱)

1 0 1 0 0 1 0 0

+1

1 0 1 0 0 1 0 1

- **نکته:** برای به دست آوردن سریع مکمل(۲) عدد کافی است عدد را به مبنای 2 برده از سمت راست عدد حرکت کرده، صفرهای احتمالی و اولین یک را تغییر نداده و بقیه بیت‌ها را نقیض کنیم.

مثال) مکمل(۲) عدد 48 را به دست آورید:

48 = 32 + 16

0 0 1 1 0 0 0 0

1 1 0 1 0 0 0 0

- **نکته:** روش مکمل(۲) روش استانداری است که برای نگهداری اعداد منفی در کامپیوتر استفاده می‌شوند.

این روش معایب روش‌های قبلی را ندارد یعنی برای صفر فقط یک نمایش داشته، ضمن آنکه از جمع یک عدد با قرینه خود صفر به دست می‌آید. استفاده از این روش موجب می‌شود تا بتوان عملیات تفریق را به کمک عملیات جمع انجام داد.

۱-۳-۲ نحوه نگهداری اعداد اعشاری

اعداد اعشاری در هر مبنایی را می‌توان به‌صورت ممیز شناور نشان داد به‌عنوان‌مثال عدد 745 را می‌توان به صورت‌های زیر نمایش داد:

745×10^0
$74/5 \times 10^1$
$7/45 \times 10^2$
$0/745 \times 10^3$
$0/0745 \times 10^4$

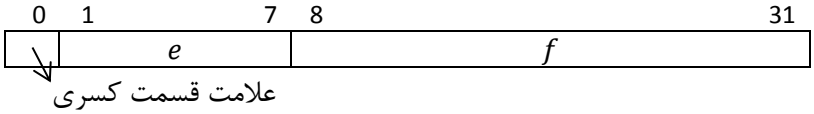
همان‌طور که ملاحظه می‌شود، ممیز جای ثابتی ندارد. به همین دلیل آن را ممیز شناور گویند.

نحوه نگهداری اعداد اعشاری از استاندارد IEEE استفاده می‌شود. مطابق این استاندارد هر عدد اعشاری را می‌توان به فرم زیر نمایش داد:

$$\pm f \times \mathrm{b}^{\pm e}$$

مقدار کسری — مبنا — توان

در این نمایش اگر $\frac{1}{b} < f < 1$ باشد می‌گوییم عدد به فرم نرمال ارائه‌شده است. فرمت استفاده‌شده در این استاندارد 32 بیتی است.



در این روش برای نمایش علامت توان از روش افزونی ۶۴ استفاده می‌شود. مطابق این روش به توان واقعی که از 64- تا 63+ تغییر می‌کند، 64 واحد اضافه می‌شود تا توان جدید به نام توان ظاهری ایجاد شود.

مثال الف) عدد 77/72- را به روش استاندارد IEEE در کامپیوتری با طول کلمات 32 بیتی نمایش دهید.

۱) تبدیل عدد از مبنای 10 به مبنای 2

$$-77/72 = -(0\,1\,0\,0\,1\,1\,0\,1\,/\,1\,0\,1\,1\,1\,0\,0\,0)_2$$

۲) تبدیل عدد از مبنای 2 به 16

$$-(01001101\,/\,10111000)_2 = -4\mathrm{D}/\mathrm{B}8 \times 16^0$$

۳) نرمال‌سازی

$$= -0/4\mathrm{DB}8 \times 16^2$$

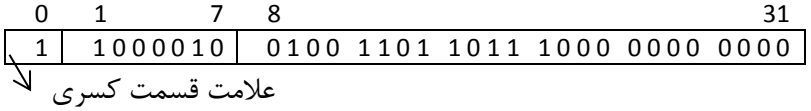
۴) استخراج مقادیر

$$\begin{cases} f = -0/4\text{DB8} \\ b = 2 \\ e = 2 + 64 = 66 = (01000010)_2 \end{cases}$$

توان واقعی توان ظاهری

۵) تولید توان ظاهری

۶) انتقال مقادیر



مثال ب) عدد 121/84- را به روش استاندارد IEEE در کامپیوتری با طول کلمات ۳۲ بیتی نمایش دهید.

۱) تبدیل عدد از مبنای 10 به مبنای 2

$$-121/84 = -(1111001 / 110101)_2$$

۲) تبدیل عدد از مبنای ۲ به ۱۶

$$-(01111001 / 11010100)_2 = -79/\text{D4}$$

۳) نرمال‌سازی

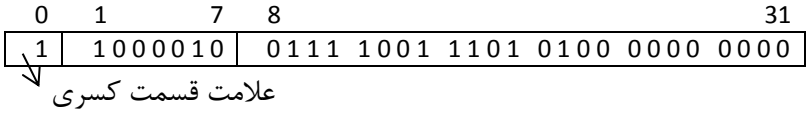
$$-79/\text{D4} = -0/79\text{D4} \times 16^2$$

۴) استخراج مقادیر

۵) تولید توان ظاهری

$$\begin{cases} f = -0/79D4 \\ b = 2 \\ e = 2 + 64 = 66 = (01000010)_2 \end{cases}$$

۶) انتقال مقادیر



۱-۴ روش‌های کدگذاری

۱) BCD ۲) EBCEDIC ۳) ASCII

۱-۴-۱ کد BCD

در روش‌های قبلی نمایش اعداد، هر عدد به‌صورت یک کمیت مستقل و قسمت نشدنی منظور شد. برای نمایش عدد به‌صورت BCD هر رقم آن به‌طور مستقل به مبنای 2 تبدیل می‌شود و هر رقم عدد، به 4 بیت تبدیل می‌گردد.

مزیت این روش در سهولت تبدیل اعداد مبنای ده به BCD و سهولت انتقال اطلاعات عددی به حافظه و از حافظه به خروجی است. ضمن آن‌که در این روش کدگذاری در صورت وقوع خطا برای یک رقم فقط آن رقم نیاز به ارسال مجدد دارد.

بدون علامت	1	5	7
	↓	↓	↓
	0001	0101	0111

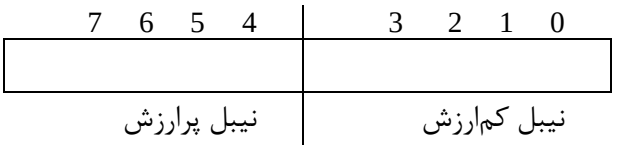
-	1	5	7
	↓	↓	
	0001	0101	0111

برای مشخص نمودن علامت عدد از جدول زیر استفاده می‌شود:

کد	علامت	کد	علامت	کد	علامت
1101	منفی	1100	مثبت	1111	بدون علامت

۱-۴-۲ کد EBCIDIC

در این روش هر رقم در 8 بیت نمایش داده می‌شوند و نیبل پردازش سمت راست‌ترین رقم علامت عدد را مشخص می‌کند. علامت‌گذاری بر اساس جدول کد BCD در نظر گرفته می‌شود.



481 بدون علامت

1111	0100	1111	1000	1111	0001		

+481

1111	0100	1111	1000	1100	0001		

-481

1111	0100	1111	1000	1101	0001		

۱-۴-۳ کد ASCII

در اوایل دوران تولید کامپیوتر، برای نمایش اطلاعات کاراکتری، کدهای مختلفی طراحی شد؛ به‌طوری‌که هر شرکت سازنده کامپیوتر، طراحی کد مربوط به خود را به کار می‌گرفت. انتخاب کد در ساخت کامپیوتر مؤثر بوده

و بعد از ساختن کامپیوتر تغییر کد به‌سادگی ممکن نبود. به دلیل عدم وجود یک استاندارد، انتقال اطلاعات از کامپیوتری به کامپیوتر دیگر، با مشکل مواجه بود. برای رفع این مشکل کمیته استاندارد تبادل اطلاعات آمریکا (ASCII) تصمیم گرفت کد استانداردی را تولید کند. این کد، کد اسکی نام گرفت.

این کد استاندارد در ابتدا 7 بیتی بوده است $(2^7 = 128)$ و پس‌ازآن، با افزایش تعداد علائم به 8 بیتی تغییریافته است $(2^8 = 256)$. با این کد، 256 کاراکتر مختلف قابل‌نمایش است.

با فشردن هر دکمه از صفحه‌کلید، سیستم معادل کد اسکی آن را به‌واسطه ورودی صفحه‌کلید ارسال می‌کند. نقل و انتقالات به مانیتور و چاپگر نیز به شکل کد اسکی انجام می‌شود.

$A \to 41H$	$a \to 61H$	$0 \to 30H$
$B \to 42H$	$b \to 62H$	$1 \to 31H$
...	...	...
$Z \to 5AH$	$z \to 7AH$	$9 \to 39H$

اختلاف کد اسکی حروف بزرگ و کوچک فقط در یک بیت است یعنی با تغییر یک بیت از عدد می‌توان حرف بزرگ را به کوچک یا کوچک را به بزرگ تبدیل کرد. (بیت پنجم)

$A \to$ 0 1 0 0 0 0 0 1 41 H
$a \to$ 0 1 1 0 0 0 0 1 61 H

۱-۵ تمرین

۱- تبدیل‌های زیر را انجام دهید.

$(39)_{10} = (?)_2$
$(25)_{10} = (?)_2$
$(29B)_H = (?)_2$

$(11001)_2 = (?)_{10}$
$(110101)_2 = (?)_{10}$
$(6B2)_H = (?)_{10}$
$(9F2D)_H = (?)_{10}$

$(45)_{10} = (?)_H$
$(629)_{10} = (?)_H$
$(1714)_{10} = (?)_H$
$(100111110101)_2 = (?)_H$

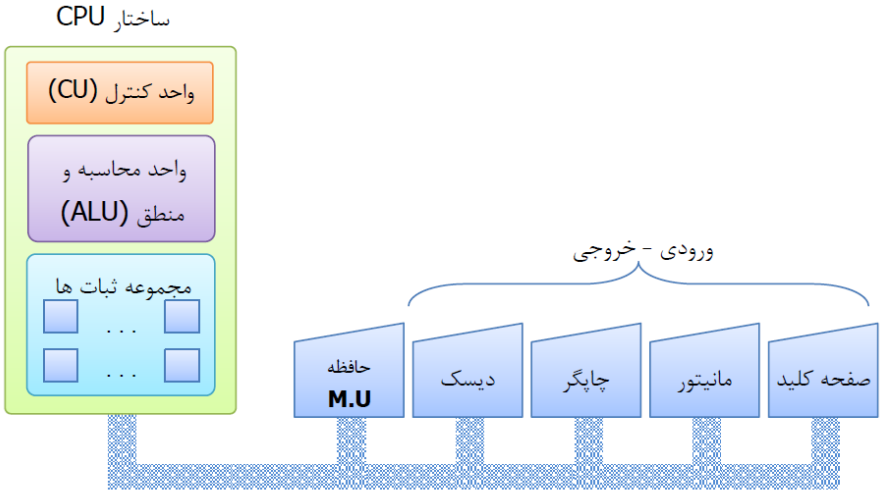
۲- حاصل تفریق دو عدد باینری 110000 و 1000 را به دست آورید.
۳- مکمل(۲) عدد 10011101 را به دست آورید.
۴- حاصل جمع دو عدد $(FF9)_H$ و $(C45)_H$ در مبنای 10 را به دست آورید.
۵- عدد $(2B8)_H$ را از عدد $(59F)_H$ کم کنید.
۶- معادل عدد 110100101000011 در کد NBCD را بنویسید.
۷- 143 اطلاعات مختلف را با چند بیت می‌توان کد کرد؟
۸- برای کد کردن حروف انگلیسی چند بیت نیاز داریم؟
۹- برای نمایش عدد دسیمال 643 چند بیت در کد BCD و چند بیت باینری لازم است؟
۱۰-عدد 97/34 را به روش استاندارد IEEE در کامپیوتری با طول کلمات ۳۲ بیتی نمایش دهید.

فصل دوم ساختمان کامپیوتر

۲-۱ ساختار کامپیوتر

کامپیوتر دارای دو قسمت نرم‌افزاری و سخت‌افزاری است. سخت‌افزار کامپیوتر شامل مدارهای الکترونیکی و الکترومکانیکی آن است و نرم‌افزار از برنامه‌های سیستم‌عامل، کامپایلرها، اسمبلرها و ... و برنامه‌های کاربران تشکیل می‌شود. برای یک کاربر که برنامه را به زبان ماشین و یا زبان اسمبلی می‌نویسد، اطلاع از طرز کار قسمت‌های مختلف و سخت‌افزار کامپیوتر بسیار ضروری است. اصولاً کامپیوتر دستگاهی است که اطلاعات دیجیتالی به‌صورت 0 و 1 را به‌عنوان ورودی می‌گیرد و بر طبق دستوراتی که در حافظه آن قرار دارند، یک سری عملیات بر روی اطلاعات مذکور انجام می‌دهد و خروجی موردنظر را تولید می‌کند. لیست دستورات را برنامه کامپیوتر نامند و محلی که این دستورات ذخیره‌شده‌اند، حافظه کامپیوتر نام دارد.

هر کامپیوتری از واحدهای ورودی، خروجی، حافظه، محاسبه و منطقی ALU [1]، باس یا گذرگاه [2] و واحد کنترل تشکیل‌شده است و مجموعه واحد محاسبه و منطقی ALU ، و واحد کنترل و ثبات‌ها [3] را، پروسسور یا CPU نامند، که وظیفه اجرای دستورات را به عهده دارد.



۲-۲ واحد پردازشگر مرکزی [4]

وظیفه پروسسور اجرای دستورات برنامه کامپیوتر است. CPU خود از واحدهای محاسبه و منطق ALU ، کنترل CU و یک سری ثبات جهت درج اطلاعات یا اعداد در آن‌ها تشکیل‌شده است. البته عملیات اصلی پردازش در واحد محاسبه و منطق ALU انجام می‌شود. به‌عنوان‌مثال فرض کنید می‌خواهیم دو عدد که در دو ثبات قرار دارند را باهم جمع کنیم. برای این کار می‌باید این دو عدد به واحد محاسبه و منطق ALU آورده شوند، که در آنجا باهم جمع گردند و نتیجه جمع ممکن است به حافظه برگردد و یا برای استفاده بعدی در ثبات‌های پروسسور ذخیره گردد.

بدیهی است عملیات ریاضی و منطقی دیگر مانند ضرب، تقسیم، مقایسه و ... نیز در واحد محاسبه و منطق ALU پروسسور انجام می‌شود.

---

[1] Arithmetic & Logical Unit (ALU)
[2] BUS
[3] Registers
[4] Central Processing Unit (CPU)

وظایف پروسسور برای اجرای هر دستور:

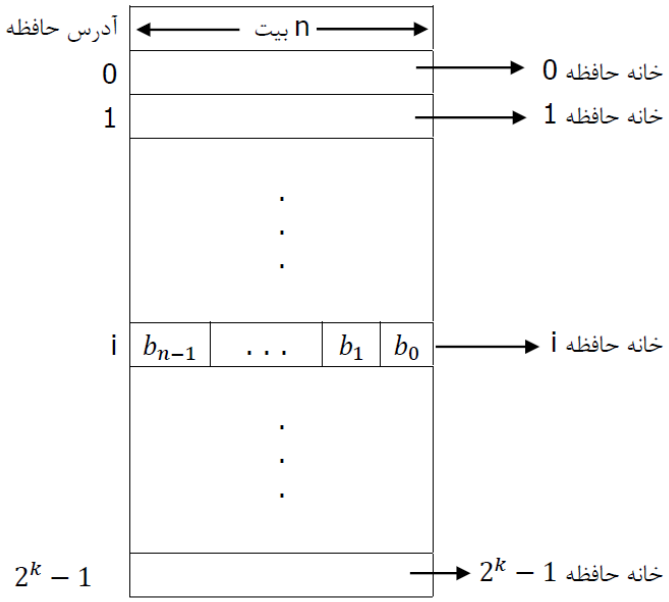
۱- واکشی[1]: منظور از عملیات واکشی انتقال دستور از حافظه به پروسسور یعنی قرار دادن در صف دستورالعمل پردازنده و بازسازی شمارنده‌ی برنامه است.

۲- رمزگشایی[2]: منظور ترجمه آدرس واکشی عملوندهای موردنیاز از حافظه است.

۳- اجرا[3]: منظور انجام محاسبات موردنیاز، ذخیره نتیجه در حافظه و یا ثبات و تغییر وضعیت پرچم‌های متصل به پردازنده است.

۲-۳ واحد حافظه[4]

حافظه کامپیوتر که حافظه اصلی نیز نامیده می‌شود، دارای یک سری سلول‌های حافظه از جنس نیمه‌هادی است که هرکدام یک بیت اطلاعات را ذخیره می‌کند، که معمولاً مجموعه‌ای از ۸ بیت این سلول‌های حافظه که به‌طور همزمان خوانده یا نوشته می‌شوند، یک بایت[5] اطلاعات یا یک‌خانه حافظه (یک بایتی) می‌نامند. البته در کامپیوترهای سریع‌تر و پرقدرت‌تر مجموعه‌ای از ۱۶ یا ۳۲ بیت یا ۶۴ بیت از سلول حافظه یک کلمه[6] یا یک‌خانه حافظه ۱۶ یا ۳۲ یا ۶۴ بیتی نامیده می‌شود. برای دستیابی به خانه‌های حافظه به هر یک از خانه‌های حافظه یک آدرس تخصیص داده می‌شود.



نکته مهم در عملیات روی حافظه، بحث آدرس آن است. به‌طورکلی دو عمل در مورد حافظه امکان‌پذیر است که در هر دو عملیات نیاز به آدرس است:

۱- عملیات خواندن

۲- عملیات نوشتن

---

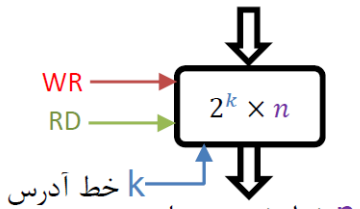
[1] Fetch
[2] DeCode
[3] Execute
[4] Memory Unit (M.U)
[5] Byte
[6] Word

اگر بخواهیم اطلاعات یک‌خانه حافظه را بخوانیم، باید آدرس آن خانه حافظه به حافظه داده شود و فرمان خواندن از واحد کنترل کامپیوتر برای حافظه ارسال گردد (پایه RD فعال شود) که در این صورت اطلاعات خانه حافظه‌ای که آدرس آن داده‌شده خوانده می‌شود.

درصورتی‌که بخواهیم اطلاعاتی در یک‌خانه حافظه نوشته شود، کافی است آدرس آن خانه حافظه به حافظه داده شود و اطلاعات مذکور نیز به حافظه ارسال گردد و فرمان نوشتن نیز از طرف واحد کنترل به حافظه فرستاده شود (پایه WR فعال شود) که اطلاعات مذکور در خانه حافظه موردنظر نوشته گردد. به‌این‌ترتیب با هر مراجعه به حافظه، می‌توان اطلاعات یک‌خانه حافظه را خواند یا در آن نوشت.

ظرفیت حافظه یکی از پارامترهای قدرت کامپیوتر است. کامپیوترهای کوچک مانند میکروکنترلرها، یا میکروپروسسورها، ممکن است چند هزار خانه حافظه یک بایتی داشته باشند، ولی کامپیوترهای متوسط و بزرگ چند صد میلیون خانه یا کلمه حافظه‌دارند.

همان‌طور که متذکر گردید حافظه برای ذخیره برنامه و اطلاعات است که این حافظه را حافظه با دستیابی تصادفی RAM[1] نامند. یعنی با دادن آدرس، می‌توان به هر یک از خانه‌های حافظه دسترسی داشت.



4 GB
$4 \times 2^{30} \times 8$
تعداد خط آدرس
$2^{32} \times 8$

1 MB
تعداد خط آدرس
$2^{20} \times 8$

۲-۴ گذرگاه[2]

همان‌طور که مشاهده شد ارتباط مداومی بین پروسسور و هر یک از دستگاه‌های ورودی، خروجی و حافظه برقرار است، برای این ارتباط راحت‌ترین وسیله ارتباط مستقیم از طریق سیم‌هایی بین آن‌ها است. ولی اگر برای ارتباط دو دستگاه باهم حدود یک‌صد سیم نیاز باشد، و اگر قرار باشد یک دستگاه مستقیماً با همه دستگاه‌های دیگر ارتباط داشته باشد، تعداد سیم‌های رابط، بسیار زیاد و غیراقتصادی می‌شود. لذا راه عملی‌تر این است که سیم‌های ارتباطی، بین عده از وسایل مشترک باشند. یعنی تعدادی از دستگاه‌ها از یک مقدار سیم، به‌طور مشترک استفاده نمایند، که این سیم‌های مشترک را Bus یا گذرگاه نامند.

تعداد سیم‌های Bus تابع نوع دستگاه‌ها است ولی در کامپیوترهای معمولی بین ۴۰ تا ۲۰۰ سیم می‌باشند. به‌این‌ترتیب پروسسور با هر واحد کامپیوتر، از طریق یک Bus ارتباط برقرار می‌کند و برای این کار لازم است پروسسور آدرس آن‌ها را روی Bus قرار دهد.

Bus خود از سه قسمت تشکیل‌شده است:

۱- Data Bus
۲- Address Bus
۳- Control Bus

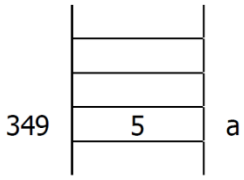
---

[1] Random Access Memory (RAM)
[2] Bus

مثال)

a = 5;
*349 ⟶ 5

$$\begin{cases} Address = 349 \\ Data = 5 \\ Control = WR \end{cases}$$



- **نکته ۱)**

عرض Data Bus حجم نقل و انتقالات بین دستگاه‌ها را نشان می‌دهد که در پردازنده 8086 برابر عرض ثبات پایه و برابر با 16 bit است.

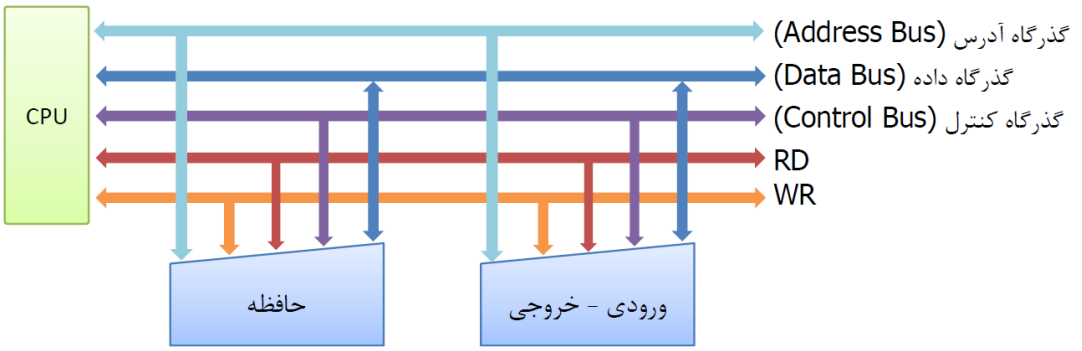
- **نکته ۲)**

عرض Address Bus تعداد بیت‌ها جهت آدرس‌دهی حافظه را نشان می‌دهد که در پردازنده 8086 برابر 20 bit است و علت آن پشتیبانی از حافظه 1 MB است.

- **نکته ۳)**

عرض Control Bus تعداد عملیات CPU روی سایر دستگاه‌ها را نشان می‌دهد که برابر 16 bit است.

با توجه به مطالب فوق، در یک سیستم کامپیوتر، انتقال اطلاعات از CPU به حافظه، یا دستگاه‌های ورودی خروجی، از طریق باس های آدرس (Address)، داده (data) و کنترل (Control) انجام می‌شوند.



۲-۵ زمان دستیابی[1]

منظور از زمان دستیابی که به‌عنوان یک پارامتر برای مقایسه حافظه‌ها استفاده می‌شود. میانگین زمان خواندن و نوشتن داده‌ها در حافظه است. قطعاً حافظه‌ای بهتر است که زمان دستیابی کوچک‌تری داشته باشد.

۲-۶ سیکل دستور[2] یا زمان اجرای دستور[3]

همان‌طوری که قبلاً ملاحظه شد، پروسسور یا CPU، دستورات را یکی‌یکی از حافظه می‌خواند و سپس اجرا می‌کند. در حقیقت زمانی برنامه شروع به اجرا می‌شود، که آدرس اولین دستور برنامه توسط اشاره‌گر دستور IP، به حافظه داده شود و فرمان خواندن نیز از طریق واحد کنترل به حافظه ارسال گردد. بعد از مدتی که برابر زمان

[1] Access Time
[2] Instruction Cycle
[3] Execution Time

دستیابی[۱] حافظه است، محتوای خانه حافظه‌ای که آدرس آن داده‌شده (در این حالت اولین دستور برنامه است) از حافظه خوانده می‌شود و وارد پروسسور می‌گردد، که این عملیات را سیکل واکشی[۲] دستور نامند. اکنون‌که دستور داخل پروسسور قرار دارد، در واحد محاسبه و منطق ALU اجرا می‌گردد، که عملیات اجرای دستور را سیکل اجرا[۳] نامند. و بالاخره مجموع سیکل واکشی و سیکل اجرا را، سیکل دستور یا زمان اجرای دستور نامند. در حقیقت زمان اجرای دستور مدتی است که یک دستور اجرا می‌شود.

$$Instruction\ Cycle = Fetch\ Cycle + Execution\ Cycle$$

به‌عنوان‌مثال در یک کامپیوتر اجرای دستور جمع دو مقدار ممکن است یک میکروثانیه طول بکشد، و در کامپیوتر سریع‌تر دستور جمع ممکن است 0.005 میکروثانیه طول بکشد. به‌این‌ترتیب یکی از پارامترهای کارایی کامپیوتر، زمان اجرای دستورات است، که به‌طور مستقیم بر روی سرعت اجرای برنامه اثر می‌گذارد.

- **تذکر**: ثباتی به نام IP [۴] (اشاره‌گر دستورالعمل) در پردازنده‌های خانواده Intel وجود داشته که همواره حاوی آدرس دستورالعمل بعدی است یعنی پروسسور به کمک این ثبات توالی اجرای دستورات برنامه را حفظ می‌کند.

۲-۷ مروری بر پردازنده‌های شرکت Intel

نام پردازنده	اندازه ثبات‌ها	پهنای باس داده	پهنای باس آدرس	Cache	حافظه	تاریخ تولید	فرکانس ساعت	تعداد ترانزیستور
8086	16 bit	20 bit	20 bit	-	1 MB	1978	5 MHz	29,000
8088	16 bit	20 bit	20 bit	-	1 MB	1979	5 MHz	29,000
80186	16 bit	20 bit	20 bit	-	1 MB	1979	5 MHz	29,000
80286	16 bit	32 bit	24 bit	-	16 MB	1982	10 MHz	134,000
80386	32 bit	32 bit	32 bit	-	4 GB	1985	16 MHz	275,000
80486	32 bit	32 bit	32 bit	8 KB	4 GB	1989	16 MHz	400,000
80586	32 bit	64 bit	32 bit	2 × 8 KB	4 GB	1993	60 MHz	3,100,000

[۱] زمانی که طول می کشد اطلاعات از حافظه خوانده یا نوشته شود، زمانی دستیابی Access Time نامیده می شود.

[2] Fetch Cycle
[3] Execution Cycle
[4] Instruction Pointer

فصل سوم معرفی پردازنده 8086

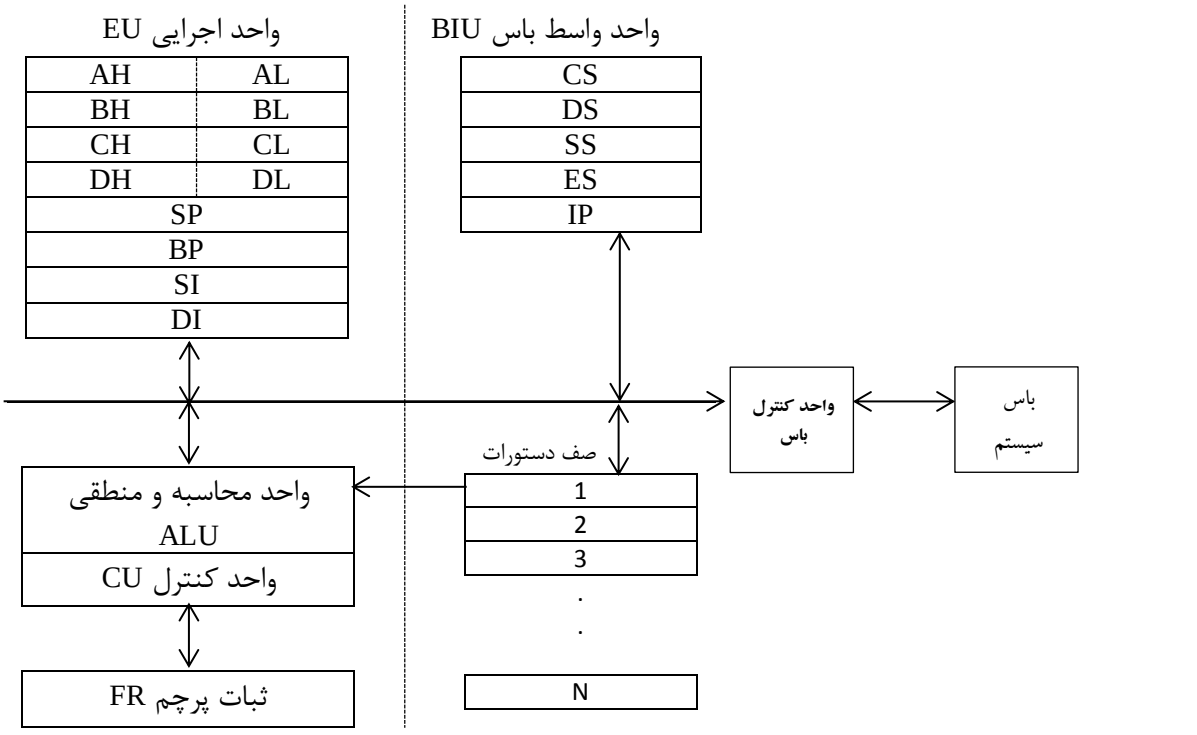
به‌منظور یادگیری زبان برنامه‌نویسی اسمبلی که یک‌زبان سطح پایین است بایستی با معماری داخلی پردازنده و نحوه ارتباط پردازنده و حافظه آشنا شد.

۳-۱ ساختار داخلی پردازنده ۸۰۸۶

پردازنده کامپیوترهای شخصی، از دو قسمت واحد اجرایی[۱] EU و واحد واسط باس[۲] BU تشکیل‌شده است. واحد اجرایی EU مسئول اجرای دستورات است، که از یک واحد محاسبه و منطقی ALU و تعدادی ثبات، تشکیل‌شده است.

واحد واسط باس BIU، شامل واحد مدیریت کنترل باس[۳]، ثبات‌های سگمنت[۴]، و صف دستورات[۵] است. یکی از کارهای مهم واحد واسط باس BIU، پیش‌خوانی[۶] دستورات از حافظه و قرار دادن آن‌ها در صف دستورات است. لذا همیشه یک سری دستورات از قبل، از حافظه خوانده می‌شوند و در صف دستورات قرار می‌گیرد، که هرلحظه واحد اجرایی بخواهد دستور را اجرا کند، بلافاصله از صف دستورات، دستور را می‌گیرد و منتظر خواندن دستور نمی‌شود. درزمانی که واحد اجرایی EU دستور مذکور را اجرا می‌کند، واحد واسط باس BIU، دستور دیگری را از حافظه می‌خواند و در صف دستورات قرار می‌دهد. به‌این‌ترتیب این دو واحد به‌صورت موازی باهم کار می‌کنند و درنتیجه سرعت CPU بالا می‌رود.

با توجه به مطالب فوق پردازنده‌های سری پنتیوم به بعد بسیاری از عملیات را به‌طور موازی انجام می‌دهند. درنتیجه سرعت آن‌ها روزبه‌روز افزایش می‌یابد.



[1] Execution Unit (EU)
[2] Bus Interface Unit (BIU)
[3] Bus Control Unit
[4] Segment Register
[5] Instruction Queue
[6] Prefetch

سخت‌افزار داخلی یک پردازنده به دو بخش عمده واحد اجرا و واحد ارتباط با باس تقسیم می‌گردد. واحد ارتباط با باس وظیفه تولید آدرس فیزیکی ۲۰ بیتی برای آدرس‌دهی خانه‌های حافظه و خواندن داده‌ها و دستورالعمل‌های برنامه از حافظه اصلی را به عهده دارد. درون هر پردازنده تعدادی رجیستر یا ثبات که وظیفه نگهداری موقت اطلاعات را به عهده‌دارند وجود دارد. در پردازنده ۸۰۸۶ رجیسترها ۱۶ بیتی هستند، پس از تولید آدرس فیزیکی ۲۰ بیتی، دستور از حافظه خوانده‌شده، از طریق باس داده وارد پردازنده گردیده، در صف دستور قرار می‌گیرد. هر دستور زبان اسمبلی، از یک سری صفر و یک تشکیل‌شده است. کنترل واحد اجرا با رمزگشایی کد دستور تشخیص خواهد داد، چه عملیاتی روی چه داده‌ای باید صورت پذیرد. بنابراین یکی از مدارات داخلی درون واحد محاسبه و منطقی (ALU[1]) را فعال می‌سازد، ورودی‌های ALU می‌توانند محتویات رجیسترها یا داده موجود روی باس داده که از حافظه خوانده‌شده، باشند. سپس پردازش لازم روی‌داده انجام‌شده، نتیجه بر روی باس داده قرار می‌گیرد که می‌تواند برای استفاده‌های بعدی درون رجیسترها ذخیره گردد. و یا از طریق باس‌داده به حافظه منتقل شود. در ضمن با توجه به نتیجه حاصل از انجام پردازش در ALU ممکن است یک یا چند بیت رجیستر FR (رجیستر پرچم) تغییر پیدا کند.

استفاده از صف، سرعت پردازنده را افزایش خواهد داد، به دلیل آنکه خواندن دستور از حافظه زمان‌بر است و بهتر است درزمانی که پردازنده مشغول اجرای یک دستور است دستورات بعدی از حافظه خوانده شوند و در صف دستور قرار گیرند. استفاده از دستورات پرش سبب کاهش سرعت می‌گردد، به دلیل اینکه سبب می‌شود از دستوراتی که درون صف قرارگرفته‌اند صرف‌نظر شود و دستورات مقصد پرش از حافظه خوانده شوند، در مدت‌زمان خواند این دستورات، واحد اجرای پردازنده بی‌کار می‌ماند. پس هرچقدر ممکن است بایستی از دستورات پرش کمتر استفاده شود.

۳-۲ ثبات‌های پردازنده

ثبات‌ها، حافظه‌های ۸، ۱۶ یا ۳۲ بیتی در داخل پردازنده مرکزی هستند. پردازنده مرکزی دارای گذرگاه داده داخلی است که پهنای آن دو برابر گذرگاه داده خارجی آن است. به‌عنوان‌مثال، برای افزایش سرعت حلقه‌های تکرار محاسبات و تصمیم‌گیری در داخل حلقه، از ثبات‌ها استفاده می‌کنیم. هر ثبات دارای نامی است که بتوان به آن مراجعه کرد. ثبات‌ها، به چند دسته تقسیم می‌شوند که عبارت‌اند از:

ثبات‌های عمومی
ثبات‌های سگمنت[2]
ثبات‌های اندیس
ثبات‌های وضعیت و کنترل

۳-۲-۱ ثبات‌ها عمومی

ثبات‌های عمومی عبارت‌اند از: AX، BX، CX و DX.

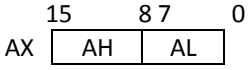
[1] Arithmatic Logic Unit
[2] Segment

بیت‌های این ثبات‌ها از سمت راست به چپ و از صفر شماره‌گذاری می‌شوند. به‌عنوان‌مثال، بیت‌های ثبات AX از صفرتا ۱۵ شماره‌گذاری می‌شوند.

۳-۲-۱-۱ ثبات انباره یا AX (ACCUMULATOR)

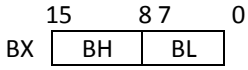
این ثبات در اعمالی که نیاز به ورودی – خروجی و محاسبات زیاد است مورد استفاده قرار میر گیرد. ثبات AX به دو بخش تقسیم می‌شود، به‌طوری‌که بایت سمت چپ را بخش بالایی و بایت سمت راست را بخش پایینی می‌نامند.

بخش بالایی تحت نام AH و بخش پایینی تحت نام AL خوانده می‌شود.



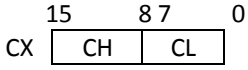
۳-۲-۱-۲ ثبات BX

این ثبات معمولاً به‌عنوان اندیسی برای توسعه آدرس مورداستفاده قرار می‌گیرد و به ثبات پایه معروف است. این ثبات در محاسبات نیز به کار گرفته می‌شود. ثبات BX نیز همانند AX به دو بخش BL و BH تقسیم می‌شود.



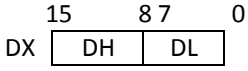
۳-۲-۱-۳ ثبات CX

این ثبات که ثبات شمارنده نامیده می‌شود، برای کنترل تعداد دفعات حلقه تکرار مورداستفاده قرار می‌گیرد. در اعمال شیفت می‌توان مقداری در آن قرارداد که تعداد شیفت را مشخص نماید. این ثبات در انجام محاسبات نیز به کار گرفته می‌شود. ثبات CX به ثبات‌های CL و CH تقسیم می‌شود.



۳-۲-۱-۴ ثبات DX

اعمال ضرب و تقسیمی که با اعداد بزرگ سروکار دارند از این ثبات استفاده می‌کنند. این ثبات در بعضی از اعمال ورودی – خروجی نیز به کار می‌رود و به ثبات داده‌ها معروف است. ثبات DX نیز به دو ثبات DL و DH تقسیم می‌شود.



دسته‌بندی ثبات‌های 8086

دسته	بیت‌ها	نام ثبات‌ها
AX ،BX ،CX ،DX	16	عمومی
AH ،AL ،BH ،BL ،CL ،DH ،DL	8	
SP (اشاره‌گر پشته)، BP (اشاره‌گر پایه)	16	اشاره‌گر
SI (اندیس مبدأ)، DI (قطعه داده)	16	اندیس
CS (قطعه کد)، DS (قطعه داده) SS (قطعه پشته)، ES (قطعه اضافی)	16	قطعه
IP (اشاره‌گر دستور)	16	دستور
FR (ثبات پرچم)	16	پرچم

۳-۳ سگمنت

قبل از بررسی ثبات‌های سگمنت، باید نگاهی مفهوم سگمنت داشته باشیم. سگمنت ناحیه‌ای از حافظه است که آدرس شروع آن بر ۱۶ قابل قسمت است و از مرز پاراگراف شروع می‌شود. اندازه سگمنت می‌تواند تا K۶۴ باشد.

هر برنامه اسمبلی می‌تواند تا چهار نوع سگمنت داشته باشد که عبارت‌اند از:

۳-۳-۱ سگمنت کد

دستورالعمل‌های زبان ماشین که باید اجرا شوند، در این سگمنت قرار می‌گیرند، به‌طوری‌که اولین دستور اجرایی برنامه در ابتدای این سگمنت قرار دارد. اگر کد برنامه بزرگ باشد (بیش از K۶۴) می‌تواند از چند سگمنت کد استفاده کند.

۳-۳-۲ سگمنت داده‌ها

داده‌ها و ناحیه‌های کار برنامه‌ها در این سگمنت قرار می‌گیرند. اگر برنامه به بیش از یک ناحیه داده نیاز داشته باشد، می‌تواند آن‌ها را تعریف و استفاده کند.

۳-۳-۳ سگمنت پشته

این سگمنت حاوی آدرس‌های برگشت از زیر برنامه است. به‌طورکلی، هر نوع اطلاعاتی که برای فراخوانی زیر برنامه‌ها لازم است در این سگمنت قرار می‌گیرد.

۳-۳-۴ سگمنت اضافی

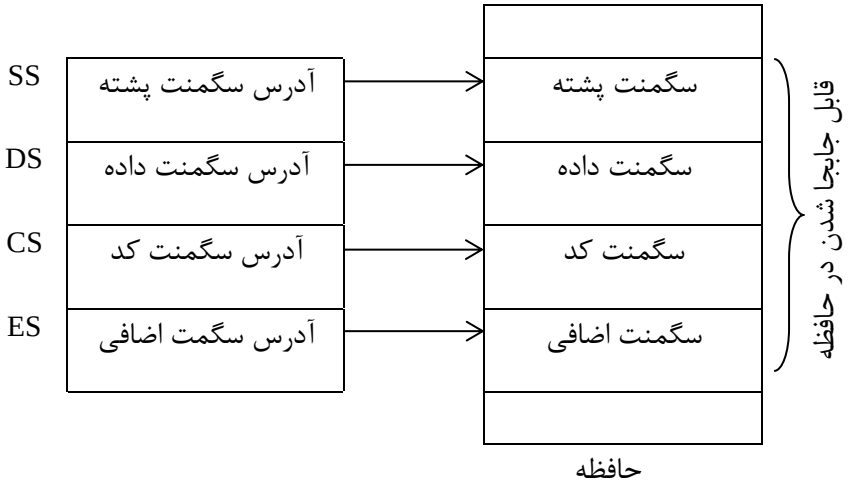
این سگمنت برای انجام عملیات بر روی رشته‌ها مورداستفاده قرار می‌گیرد و در این اعمال، برای مدیریت آدرس‌دهی حافظه به کار می‌رود.

۳-۳-۵ ثبات‌های سگمنت

هر ثبات سگمنت، آدرس شروع یک سگمنت را در خودش نگهداری می‌کند.

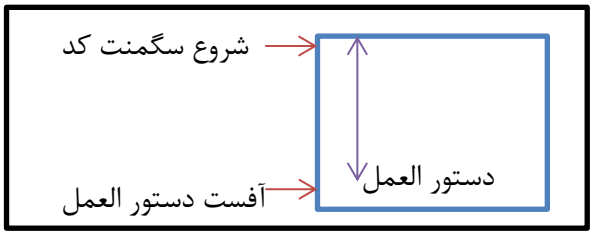
ثبات‌های سگمنت عبارت‌اند از: CS، DS، SS و ES که هرکدام ۱۶ بیتی هستند.

- ثبات CS (ثبات سگمنت کد)، حاوی آدرس شروع سگمنت کد برنامه است که در آدرس‌دهی دستورات مورداستفاده قرار می‌گیرد.
- ثبات DS (ثبات سگمنت داده‌ها)، حاوی آدرس شروع ناحیه داده‌ها را در خودش نگهداری می‌کند، به‌طوری‌که، دستورالعمل‌های برنامه، برای مراجعه به داده‌ها از این آدرس استفاده می‌کنند.
- ثبات SS (ثبات سگمنت پشته)، آدرس شروع سگمنت را نگهداری می‌کند. این ثبات معمولاً توسط سیستم مورداستفاده قرار می‌گیرد و برنامه‌نویس کمتر به آن مراجعه می‌نماید.
- ثبات ES (ثبات سگمنت اضافی)، آدرس شروع سگمنت اضافی را نگهداری می‌کند.



۳-۳-۶ ثبات‌های اندیس

ثبات‌های اندیس حاوی آفست داده‌ها و دستورالعمل‌ها در داخل سگمنت ها هستند. منظور از آفست، فاصله متغیر، برچسب یا دستورالعمل از ابتدای سگمنت آن است. ثبات‌های اندیس در پردازش رشته‌ها، آرایه‌ها و سایر ساختمان داده‌هایی که حاوی چند عنصر هستند، موجب افزایش سرعت می‌شوند. ثبات‌های اندیس عبارت‌اند از: BP، SP، SI و DI.



۳-۳-۶-۱ ثبات BP

این ثبات حاوی آفستی از ثبات SS (ثبات پشته) است. در فراخوانی زیر برنامه‌ها (که در آینده با آن‌ها آشنا خواهید شد)، چنانچه پارامترها از طریق پشته به زیر برنامه منتقل شوند، از طریق ثبات BP قابل بازیابی خواهند بود.

۳-۳-۶-۲ ثبات SP

ثبات SP حاوی آفست بالای پشته است، ثبات‌های SP و SS باهم ترکیب می‌شوند تا آدرس کامل بالای پشته را ایجاد کنند.

۳-۳-۶-۳ ثبات SI

این ثبات برای عملیات رشته‌ای مورداستفاده قرار می‌گیرد و آدرس رشته منبع را نگهداری می‌کند. به همین دلیل آن را ثبات اندیس منبع گویند.

۳-۳-۶-۴ ثبات DI

ثبات DI آدرس رشته مقصد را در عملیات رشته‌ای نگهداری می‌کند و به همین دلیل ثبات اندیس مقصد نام دارد.

۳-۳-۷ ثبات‌های وضعیت و کنترل

۳-۳-۷-۱ ثبات IP

این ثبات همواره حاوی آفست دستور اجرایی بعدی در سگمنت کد است. ثبات‌های IP و CS برای تعیین آدرس دستور بعدی به کار می‌روند.

۳-۳-۷-۲ ثبات فلگ ها[1]

ثبات مخصوصی است که بیت‌های آن، وضعیت پردازنده مرکزی یا نتیجه عملیات محاسباتی را نشان می‌دهد. هر بیت دارای نامی است و تعدادی از این بیت‌ها نیز بلااستفاده‌اند.

۹ بیت از ۱۶ بیت ثبات فلگ مورداستفاده قرار می‌گیرد و در حالت فعلی کامپیوتر و نتایج حاصل از پردازش را مشخص می‌کند.

۱۵	۱۴	۱۳	۱۲	۱۱	۱۰	۹	۸	۷	۶	۵	۴	۳	۲	۱	۰
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

[1] Flags

بیت CF

C مخفف Carry به معنی رقم نقلی است و این بیت حاوی رقم نقلی از آخرین بیت در انجام محاسبات و یا شیفت است.

(F به معنی Flag است)

بیت DF

D مخفف Direction و به معنی جهت است و این بیت برای کنترل اعمال رشته‌ای مثل مقایسه یا انتقال رشته‌هایی که بیش از یک کلمه‌اند، به کار می‌رود. اگر این بیت برابر با یک باشد، عمل مقایسه یا شیفت از سمت راست به چپ وگرنه عمل مقایسه یا شیفت از چپ به راست انجام می‌شود.

بیت PF

P مخفف Parity و به معنی توازن است و این بیت برای کنترل صحت اطلاعات به کار می‌رود. اگر این بیت برابر با یک باشد، بیانگر این است که تعداد بیت‌های شیفت داده‌شده زوج است و اگر صفر باشد، بیانگر این است که تعداد بیت‌های شیفت داده‌شده فرد است.

بیت AF

A مخفف Auxiliary Carry و به معنی رقم نقلی کمکی است. چنانچه در محاسبات ۸ بیتی، رقم نقلی در بیت سوم ایجاد شود، این بیت برابر با یک خواهد شد.

بیت ZF

Z مخفف Zero به معنی صفر است و چنانچه نتیجه اعمال محاسبات یا مقایسه برابر با صفر باشد، این بیت برابر با یک خواهد شد.

بیت SF

S مخفف Sign به معنی علامت است و برای بررسی نتیجه عملیات محاسباتی به کار می‌رود. اگر نتیجه عملیات منفی باشد این بیت برابر یک وگرنه برابر با صفر است.

بیت T

T مخفف Trap و به معنی قدم‌به‌قدم است. چنانچه این بیت برابر با یک باشد، اجرای برنامه به‌صورت دستور به دستور انجام می‌شود.

بیت I

I به معنی Interrupt و به معنی وقفه است. اگر این بیت یک باشد سیستم به وقفه‌ها پاسخ می‌دهد وگرنه وقفه‌ها را نادیده می‌گیرد.

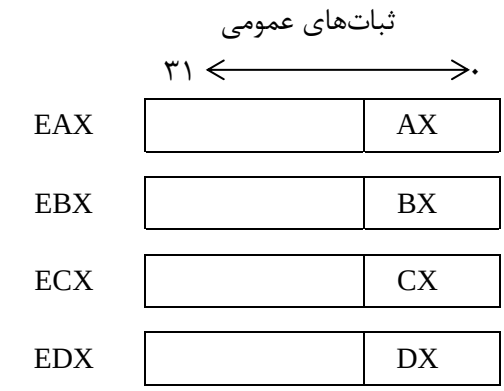
بیت O

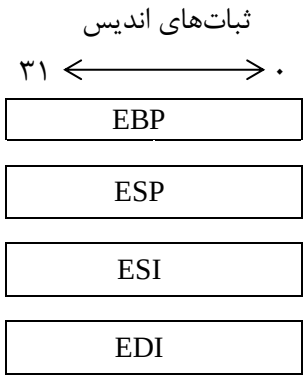
O مخفف Ovreflow و به معنی سرریز است. چنانچه در انجام محاسبات، آخرین بیت(بیت باارزش) به دلیل سرریز شدن از بین برود. بیت O برابر صفر خواهد شد.

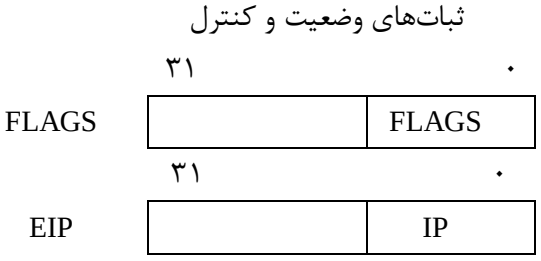
۳-۳-۸ ثبات‌های ۳۲ بیتی

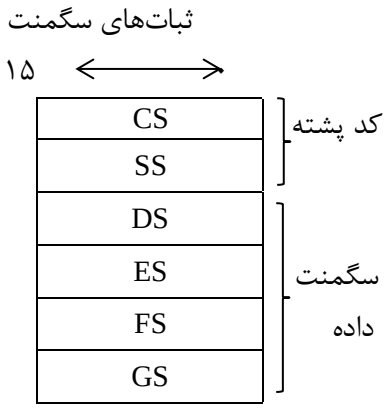
تمام پردازنده‌های اینتل از 80386 به بعد، ثبات‌های ۳۲ بیتی دارند. با استفاده از ثبات‌های ۳۲ بیتی، به فضای آدرس زیادی می‌توان دست‌یافت. در این پردازنده‌ها، ثبات‌های سگمنت ۱۶ بیتی‌اند، اما توجه داشته باشید فضای آدرس زیادی می‌توان دست‌یافت.

در این پردازنده‌ها، ثبات‌های سگمنت ۱۶ بیتی‌اند، اما توجه داشته باشید که دو ثبات جدید به نام FS و GS به این پردازنده‌ها اضافه شدند. قسمت بالایی ثبات‌های عمومی ۳۲ بیتی، نام‌گذاری نشده‌اند. برای نوشتن مقدار ۱۶ بیتی در نیمه بالایی این ثبات‌ها، ابتدا باید آن را در نیمه پایینی قرار دهید، سپس آن را به سمت چپ شیفت دهید. نکته قابل‌توجه این است که سرعت پردازنده از سرعت حافظه بیشتر است و پردازنده باید منتظر بماند تا حافظه دستورالعمل را ارسال نماید. پردازنده‌های پیشرفته از تکنیک‌های زیر استفاده می‌کنند تا دچار مشکل نشوند.









۳-۴ حافظه RAM

حافظه بخشی از جنبه سخت‌افزاری کامپیوتر است که اطلاعات را نگهداری می‌کند و تقسیمات خاص خودش را دارد. در زیر، واحدهای حافظه که در زبان اسمبلی می‌توان از آن‌ها استفاده کرد و به آن‌ها دستیابی داشت، آمده است:

- کوچک‌ترین واحد حافظه که می‌تواند صفر یا یک را نگهداری کند. بیت نام دارد.
- مجموعه‌ای از ۸ بیت که می‌تواند یک حرف را نگهداری نماید، بایت نام دارد.
- مجموعه‌ای از دو بایت، کلمه نام دارد.
- کلمه مضاعف مجموعه‌ای از ۴ بایت است.
- چهارکلمه‌ای مجموعه‌ای از ۸ بایت است.
- پاراگراف ناحیه‌ای ۱۶ بایتی است.

در حافظه کامپیوتر، بایت‌ها از صفر شماره‌گذاری می‌شوند و شماره‌های هر بایت را آدرس آن بایت گویند. بنابراین، آدرس اولین بایت برابر با صفر، دومین بایت برابر با یک و آدرس n امین بایت برابر با n-1 است.

۳-۵ حافظه ROM

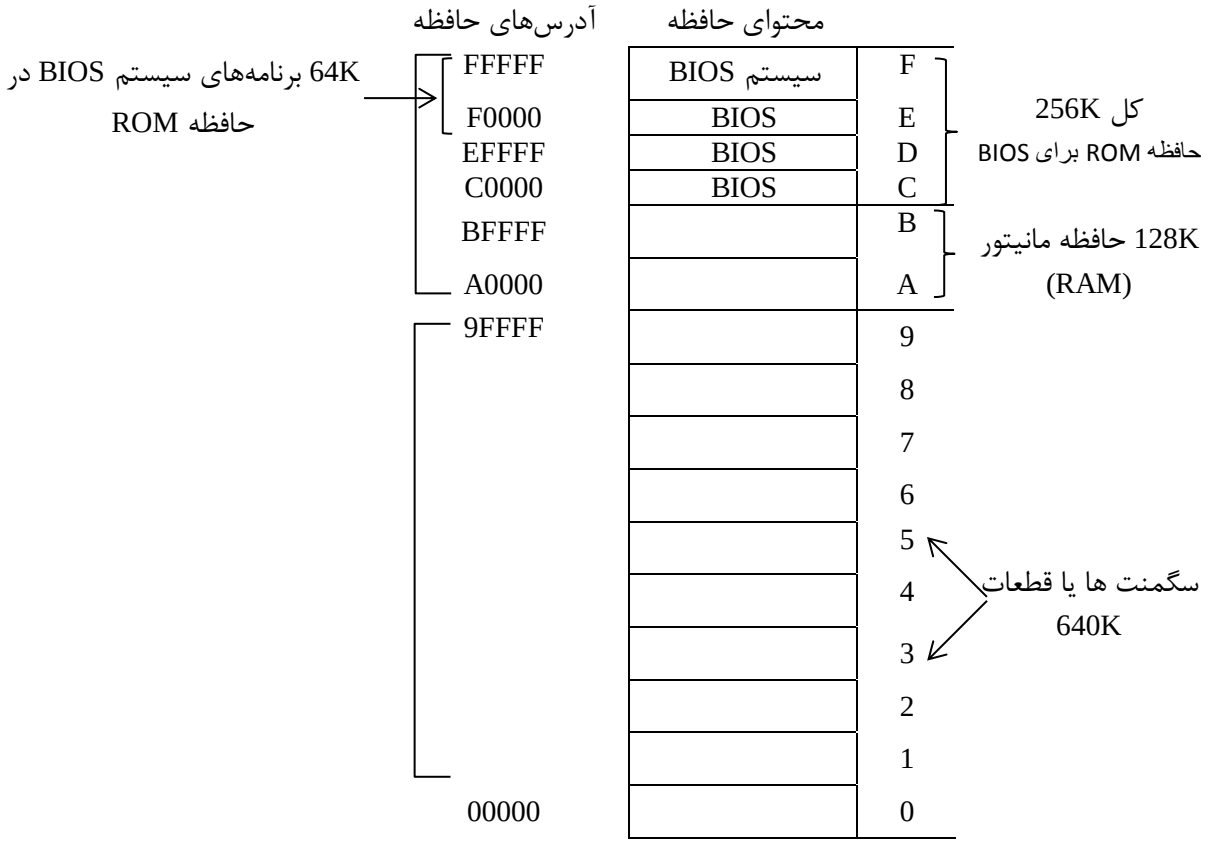
ROM BIOS، تشکیل‌دهنده اصلی سیستم‌عامل کامپیوتر است. بایاس حاوی نرم‌افزار پیکربندی و تشخیص مسیر و روال‌های ورودی – خروجی سطح پایین است که توسط DOS مورداستفاده قرار می‌گیرد. بایاس، در تراشه ریزپردازنده در مادر بورد سیستم قرار دارد و توسط کارخانه سازنده، برنامه‌نویسی می‌شود. بسیاری از سیستم‌ها از مشخصات بایاس استانداردی پیروی می‌کنند که توسط شرکت‌هایی مثل فونیکس یا AMI (بایاس اِوارد) ساخته می‌شوند.

برنامه‌های موجود در ROM را میان‌افزار[1] می‌گویند، زیرا نرم‌افزاری است که در رسانه سخت‌افزاری ذخیره‌شده است. بایاس در بالاترین آدرسی که توسط پردازنده ۸۰۸۶ یا ۸۰۸۸ قابل‌دستیابی است قرار دارد.

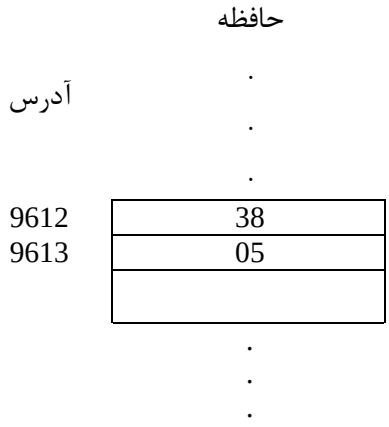
حافظه کامپیوترهای شخصی در یک مگابایت اول است و از آدرس صفر 00000 تا آدرس 9FFFFH (مقدار 640K)، حافظه RAM است که در اختیار کاربر است. از آدرس A0000H تا BFFFFH یعنی از 640K تا 768K (128K حافظه RAM است) برای عملیات مانیتور است و از آدرس C0000H تا EFFFFH یعنی از 768K تا

[1] firmware

960K به حافظه ROM اختصاص دارد و برای روتین‌هایی ورودی خروجی پایه BIOS[1]، که جهت سرویس دستگاه‌های ورودی خروجی، مانند دیسک، چاپگر و ... بکار می‌روند و بالاخره از آدرس F0000H تا FFFFFH مقدار 64K حافظه است که برای برنامه‌های سیستم می‌باشند و جهت تست کردن قسمت‌های مختلف کامپیوتر، روتین‌های BIOS و برنامه بوتینگ[2] یا برنامه راه‌اندازی ... از آن استفاده می‌کنند.



اگر اطلاعاتی که در داخل حافظه باید ذخیره شود، ۱۶ بیتی باشد مانند عدد 0538H در این صورت یک بایت کم‌ارزش‌تر عدد یعنی عدد 38 در خانه حافظه بعدی ذخیره می‌گردد. به‌عنوان‌مثال 38 در خانه حافظه با آدرس 9612 ذخیره می‌شود و 05 در خانه حافظه با آدرس 9613 ذخیره می‌گردد.



[1] Basic Input Output System (BIOS)
[2] Booting Program

برای اینکه یک عدد در مبنای 16، بر 16 قابل قسمت باشد بایستی رقم سمت راست آن حتماً صفر باشد. در یک عدد پنج‌رقمی مانند $a_4\ a_3\ a_2\ a_1\ a_0$ در مبنای 16 رقم‌های $a_4\ a_3\ a_2\ a_1$ همه مضارب 16 هستند، درصورتی‌که a_0برابر صفر باشد کل عدد مضرب 16 خواهد بود، در غیر این صورت فاکتور 16 از عدد به دست نمی‌آید و بنابراین بر 16 بخش‌پذیر نیست.

با توجه به 16 بیتی بودن رجیسترهای پردازنده و 20 بیتی بودن آدرس خانه‌های حافظه، می‌توان از رقم صفر سمت راست آدرس ابتدای سگمنت صرف‌نظر نموده و فقط 4 رقم سمت چپ آدرس ابتدای سگمنت را در رجیستر سگمنت مربوط به هر سگمنت قرارداد. (CS رجیستر سگمنت مربوط به سگمنت کد، DS رجیستر سگمنت مربوط به سگمنت داده و SS رجیستر سگمنت مربوط به سگمنت پشته و ES رجیستر سگمنت مربوط به سگمنت اضافی.)

۳-۵-۱ انواع آدرس

۳-۵-۱-۱ آدرس مطلق یا فیزیکی

آدرس 20 بیتی که روی باس آدرس قرارگرفته، خانه‌های حافظه را آدرس‌دهی می‌نماید. محدوده تغییر این آدرس از $(00000)_{16}$ تا (FFFFF) است.

۳-۵-۱-۲ آدرس آفست

آدرس نسبی و 16 بیتی نسبت به ابتدای یک سگمنت، با توجه به اینکه ظرفیت یک سگمنت از 0 تا 65535 تغییر می‌کند. با توجه به‌تساوی $2^{16}-1=(1111111111111111)_2=(FFFF)_{16}$ آدرس آفست اولین خانه یک سگمنت $(0000)_{16}=(0000000000000000)_2$ و آدرس آفست آخرین خانه یک سگمنت $(FFFF)_{16}=(1111111111111111)_2$ خواهد بود.

۳-۵-۱-۳ آدرس منطقی

ترکیبی از دو آدرس 16 بیتی به‌صورت:

مقدار آفست: محتوای رجیستر سگمنت است. برای تبدیل آدرس منطقی به فیزیکی، محتوای رجیستر سگمنت، چهار بیت به سمت چپ شیف داده می‌شود و از سمت راست چهار صفر مبنای دو وارد می‌گردد. سپس این مقدار با آدرس آفست جمع شده، آدرس فیزیکی 20 بیتی تولید می‌شود، این کار توسط واحد شیفت دهنده و جمع کننده انجام می‌گردد.

برای مثال فرض کنید، آدرس ابتدای سگمنت داده $(14560)_{16}$ باشد، بنابراین محتوای رجیستر DS برابر $(1456)_{16}$ خواهد بود، با توجه به مطالب گفته‌شده، آدرس منطقی اولین خانه این سگمنت 1456H:0000H و آدرس منطقی آخرین خانه آن 1456H:FFFFH است. برای به دست آوردن آدرس فیزیکی، 1456H چهار بیت به سمت چپ شیف داده می‌شود و چهار بیت صفر از سمت راست وارد می‌شود، با توجه به اینکه هر چهار بیت

مبنای دو معادل یک رقم مبنای شانزده است، این چهار بیت صفر، یک صفر مبنای شانزده جلوی عدد ظاهر می‌کنند. بنابراین آدرس فیزیکی اولین و آخرین خانه سگمنت به شرح زیر است:

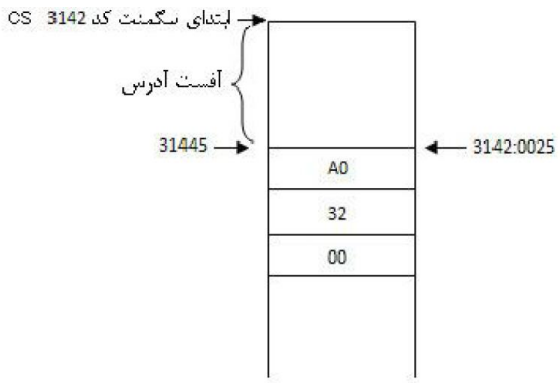
$$1456H = (0001010001010110)_2 \left(\frac{0001}{1}\frac{0100}{4}\frac{0101}{5}\frac{0110}{6}\frac{0000}{0}\right)_2 = 14560H$$

آدرس فیزیکی آخرین خانه:
$$\begin{array}{r} 14560H \\ +FFFFH \\ \hline 2455FH \end{array}$$

آدرس فیزیکی اولین خانه:
$$\begin{array}{r} 14560H \\ +0000H \\ \hline 14560H \end{array}$$

مجموعه ثبات‌های CS:IP را آدرس منطقی دستور می‌نامند.

به‌عنوان‌مثال CS:IP= ۳۱۴۲:۰۰۲۵ بدین معناست که آدرس شروع سگمنت کد ۳۱۴۲۰ و آفست آدرس ۰۰۲۵ است. اگر در این آدرس اطلاعات 0032H را ذخیره کنیم، نحوه ذخیره‌سازی اطلاعات در حافظه به‌صورت زیر است:



آدرس واقعی
$$\begin{array}{r} 31420 \\ +0025 \\ \hline 31445 \end{array}$$

فصل چهارم ساختار برنامه اسمبلی

گرچه CPU به دودویی کار می‌کند ولی می‌تواند اعمال را با سرعت بسیار بالایی انجام دهد. با این‌وجود برای انسان برنامه‌نویسی یک کامپیوتر با 0 و 1 کاری طاقت‌فرسا است. برنامه‌ای که از 0ها و 1ها ساخته‌شده باشد زبان ماشین نام دارد و روزگار اولیه پیدایش کامپیوتر، برنامه‌نویس‌ها درواقع برنامه‌ها را به زبان ماشین می‌نوشتند. گرچه سیستم مبنای شانزده به‌عنوان راه کاراتری برای نمایش اعداد دودویی بکار رفت، فرآیند کار با کد ماشین هنوز برای انسان خسته‌کننده بود. نهایتاً زبان‌های اسمبلی به وجود آمده‌اند و برای دستورات کد ماشین، نماد رمزی (خلاصه) راه همراه با امکانات دیگری که برنامه‌نویسی را سریع و کم خطا می‌نمود، فراهم کردند. کلمه نماد رمزی بارها در مقالات علوم و مهندسی کامپیوتر برای اشاره به کدها و خلاصه‌سازی، برای به خاطر سپردن آن‌ها به‌کاررفته است. برنامه‌های اسمبلی نوشته‌شده در این سبک باید به‌وسیله برنامه‌ای بنام اسمبلر به کد ماشین ترجمه شوند. زبان اسمبلی به یک‌زبان سطح پایین معروف است زیرا مستقیماً با ساختار درونی CPU سروکار دارد. برای برنامه‌نویسی به زبان اسمبلی برنامه‌نویس باید تعداد ثبات‌ها و سایز آن‌ها همراه با جزئیات CPU را بداند. امروزه هرکس می‌تواند زبان‌های برنامه‌نویسی متعددی، همچون Pascal، Basic، C و نظایر آن‌ها را به کاربرد. این برنامه‌ها را زبان‌های سطح بالا می‌گویند زیرا برنامه‌نویس نیازی به دانستن جزئیات درون CPU ندارد. درحالی‌که برای ترجمه یک برنامه زبان اسمبلی به کد ماشین، از اسمبلر استفاده می‌شود، (گاهی آن را کد مقصود می‌نامند)، زبان‌های سطح بالا به‌وسیله کامپایلر به کد ماشین تبدیل می‌شوند. مثلاً برای نوشتن برنامه‌ای در C، باید از کامپایلر C برای ترجمه برنامه به زبان ماشین استفاده کرد.

برنامه‌نویسی به زبان‌های سطح بالا بسیار ساده‌تر از برنامه‌نویسی به زبان‌های سطح پایین، مثل اسمبلی است. دستورات زبان‌های سطح بالا به زبان محاوره‌ای نزدیک هستند و هنگام ترجمه، هر دستور زبان سطح بالا ممکن است به ده‌ها دستور زبان ماشین اسمبلی تبدیل شود. اما برنامه‌نویسی به زبان اسمبلی درعین‌حال که دشوار است، امتیازاتی دارد که عبارت‌اند از:

۱- برنامه‌های زبان اسمبلی نسبت به برنامه‌های زبان‌های سطح بالا حافظه کمتری را اشغال می‌کنند و سرعت اجرای آن‌ها نیز بالا است.
۲- برنامه‌نویس در زبان اسمبلی، کارهای فوق‌العاده‌ای را می‌تواند انجام دهد، به‌طوری‌که بعضی از این کارها در زبان‌های سطح بالا امکان‌پذیر نیست.
۳- برنامه‌های که نیاز به زمان بلادرنگ دارند، در زبان اسمبلی به‌خوبی نوشته می‌شوند.
۴- برنامه‌های مقیم در حافظه به زبان اسمبلی راحت‌تر نوشته می‌شوند.

۱-۴ ملزومات زبان اسمبلی

برای اینکه بتوانید به زبان اسمبلی برنامه‌نویسی کنید باید امکانات زیر را داشته باشید:

۱- کامپیوتر شخصی همساز با IBM
۲- نسخه‌ای از سیستم‌عامل DOS و آشنایی با فرامین آن
۳- ویراستاری که برنامه اسمبلی را در آن تایپ کنید. می‌توانید از ویراستار Edit که همراه سیستم‌عامل DOS وجود دارد بهره ببرید (هر ویراستار متنی دیگر را می‌توانید مورداستفاده قرار دهید).
۴- چون برنامه‌های اسمبلی نیز مانند برنامه‌های سطح بالا به زبان ماشین ترجمه می‌شوند، باید مترجم زبان اسمبلی که اسمبلر نامیده می‌شود در اختیارتان باشد. از توربو اسمبلر یا ماکرو اسمبلر استفاده کنید.
۵- آشنایی با سیستم اعداد و ساختمان ماشین

۲-۴ شناسه

شناسه[1]، از عناصر برنامه زبان اسمبلی است. مثل نام برنامه یا زیر برنامه، برچسب‌ها و عملوندها. هر شناسه دارای نام باشد و برای نام‌گذاری شناسه از ترکیبی از حروف a تا z و A تا Z (تفاوتی بین حروف کوچک و بزرگ نیست)، ارقام صفرتا ۹ و کاراکترهایی مثل ?، - ، $ ، . ، @ استفاده می‌شود، به‌طوری‌که با ارقام و نقطه شروع نشود. چون اسمبلرها از نمادهای خاصی که با @ شروع می‌شود، استفاده می‌کنند، برای نام‌گذاری شناسه‌ها از @ استفاده نکنید. حداکثر طول نام هر شناسه، ۳۱ کاراکتر است. بعضی از شناسه‌های مجاز عبارت‌اند از: price، total، $P12 و t_1.

۳-۴ قالب کلی دستورات

هر برنامه اسمبلی، مجموعه‌ای از چند دستور است. بعضی از دستورات اجرایی هستند و باید به زبان ماشین ترجمه شوند و بعضی دیگر راهنمای اسمبلر هستند و به اسمبلر می‌گویند که کار خاصی مثل تعریف داده را انجام دهد. قالب کلی دستورات اسمبلی به‌صورت زیر است:

[توضیحات ;] [عملوندها] دستورالعمل [شناسه]

در شکل کلی کاربرد دستورات، شناسه، عملوندها و توضیحات در داخل کروشه ([ ]) قرار دارند و معنایش این است که اختیاری‌اند. به‌عبارت‌دیگر، دستورالعمل اسمبلی ممکن است فاقد شناسه، عملوند و توضیحات باشد.

هرکدام از اجزای تشکیل‌دهنده شکل کلی دستورالعمل‌های اسمبلی، حداقل باید با یک‌فاصله از هم جدا شوند، ولی بهتر است هرکدام از این اجزا را از ستون خاصی شروع کنید تا خوانایی برنامه بالا باشد. به‌عنوان‌مثال، شناسه‌ها را از ستون ۱، دستورالعمل را از ستون ۱۵، عملوندها را از ستون ۳۰ و توضیحات را بعد از عملوندها تایپ کنید.

شناسه می‌تواند آدرس یک‌قلم داده یا آدرس یک دستورالعمل، زیر برنامه یا سگمنت باشد. وجود شناسه‌ها در برنامه موجب می‌شود تا برنامه‌نویس بتواند به اقلام داده یا دستورالعمل، زیر برنامه و سگمنت مراجعه کند.

دستورالعمل مشخص می‌کند که چه‌کاری باید توسط کامپیوتر صورت گیرد. دستورالعمل‌های زبان اسمبلی به دودسته تقسیم می‌شود:

دستوراتی که به کد زبان ماشین ترجمه می‌شوند.

دستوراتی که راهنمای اسمبلر نام دارند و به اسمبلر می‌گویند که چه‌کاری انجام دهد. این دستورات به زبان ماشین ترجمه نمی‌شوند. هر دو نوع دستورالعمل در ادامه تشریح می‌شوند.

هر دستورالعمل زبان اسمبلی می‌تواند یک، دو یا هیچ عملوند داشته باشد. اگر دستورالعملی دو عملوند داشته باشد، به‌صورت زیر استفاده می‌گردد:

[توضیحات ;] <عملوند> و < عملوند مقصد > دستورالعمل [شناسه]

عملوند مقصد، جایی است که نتیجه عمل دستورالعمل باید در آنجا قرار گیرد و می‌تواند ثبات یا یک محل حافظه (آدرس حافظه) باشد و عملوند منبع جایی است که اطلاعات موردنیاز دستورالعمل در آنجا قرار دارد. عملوند منبع نیز می‌تواند یک مقدار ثابت، ثبات یا یک محل حافظه باشد.

[1] Indentifire

ازآنجایی‌که درک عملکرد برنامه زبان اسمبلی، کمی دشوار است، لذا بهتر است دستورالعمل‌ها به همراه توضیحات نوشته شوند. توضیحات می‌تواند در هر جای برنامه (در انتهای دستور، یا در یک سطر جداگانه) قرار داشته باشند. توجه کنید که توضیحات باید با ; شروع شوند.

۴-۴ قالب برنامه اسمبلی

در زبان اسمبلی، برنامه دارای شکل خاصی است که نمونه ساده‌ای از آن در شکل زیر آمده است.

```
                    تعریف سگمنت پشته
                    تعریف سگمنت داده
نام سگمنت کد        Segment
نام برنامه          Proc far
                    .
                    .
                    .
                    .
نام برنامه          Ednp
                    Ends
نام سگمنت کد        End          نام برنامه
```

هر سگمنت باید دارای نام باشد و نام سگمنت یک شناسه است. بعضی از اسامی مجاز سگمنت ها عبارت‌اند از: codsg، stsg، datsg. همان‌طور که در شکل مشاهده می‌شود، تعریف سگمنت ها از اهمیت ویژه‌ای برخوردار است.

۴-۵ تعریف سگمنت ها

همان‌طور که قبلاً گفته شد، هر برنامه ممکن است چند سگمنت داشته باشد، مثل سگمنت داده‌ها، سگمنت پشته و سگمنت کد، برای تعریف هر سگمنت، از راهنمای segment به‌صورت زیر استفاده می‌شود:

```
نام سگمنت segment پارامترها

....

نام سگمنت Ends
```

نام سگمنت مانند یک شناسه انتخاب می‌شود. دستور SEGMENT شروع سگمنت و دستور ENDS انتهای سگمنت را مشخص می‌کند و سه‌نقطه (...) عملِ تعریف سگمنت را انجام می‌دهد. به‌عنوان‌مثال، در سگمنت داده‌ها، داده‌ها را مشخص می‌کند و در سگمنت کد، دستورالعمل‌های برنامه را تعیین می‌کند. دقت کنید که نام سگمنت، هم قبل از SEGMENT و هم قبل از ENDS آمده است.

پارامترهایی که در دستور SEGMENT به کار می‌روند، بر سه نوع‌اند: پارامتر تنظیم[1]، پارامتر ترکیب[2] و پارامتر کلاس[3]. ترتیب قرار گرفتن این سه پارامتر به‌صورت زیر است:

[پارامتر کلاس] [پارامتر ترکیب] [پارامتر تنظیم] segment نام سگمنت

۴-۵-۱ پارامتر تنظیم

مرزی را که سگمنت باید ازآنجا شروع شود مشخص می‌کند. پارامتر تنظیم می‌تواند یکی از مقادیر BYTE، PARA،WORD یا PAGE را بپذیرد. معنای هرکدام از این پارامترها به‌صورت زیر است:

- BYTE: آدرس سگمنت می‌تواند از هر نقطه‌ای از حافظه شروع شود.
- WORD: آدرس سگمنت می‌تواند از هر نقطه‌ای از حافظه که آدرس آن زوج باشد شروع شود.
- PARA: آدرس سگمنت از مرز پاراگراف (جایی که بر ۱۶ قابل قسمت باشد) شروع می‌شود.
- PAGE: آدرس سگمنت می‌تواند از هرجایی که بر ۲۵۶ قابل قسمت باشد شروع شود.

اگر پارامتر تنظیم ذکر نشود سیستم به‌طور خودکار آن را PARA در نظر می‌گیرد.

۴-۵-۲ پارامتر ترکیب

پارامتر ترکیب مشخص می‌کند که آیا این سگمنت با سگمنت های دیگری که پس از ترجمه برنامه به آن پیوند داده می‌شوند، ترکیب شود یا خیر. مقادیر آن NONE، PUBLIC، STACK، COMMON و AT است.

- NONE: سگمنت به‌طور منطقی از سگمنت های دیگر جدا است، ولی ممکن است به‌طور فیزیکی در کنار هم باشند. در این صورت فرض می‌شود هر سگمنت آدرس پایه مخصوص به خود را دارد.
- PUBLIC: موجب می‌شود که برنامه پیونددهنده سگمنت های PUBLIC بانام و کلاس یکسان را در کنار هم قرار دهد. برای تمام این سگمنت ها یک آدرس در نظر گرفته می‌شود.
- STACK: برنامه پیونددهنده پشته را عمومی در نظر می‌گیرد. حداقل یک پشته در نظر گرفته می‌شود. اگرچند پشته وجود داشته باشد، SP به اولین پشته اشاره خواهد کرد.
- COMMON: موجب می‌شود برنامه پیونددهنده به سگمنت های بانام و کلاس یکسان، آدرس یکسانی بدهد. در حین اجرا، دومین سگمنت بر روی اولین سگمنت قرار می‌گیرد. طول ناحیه اشتراکی توسط بزرگ‌ترین سگمنت مشخص می‌شود.
- AT: این پارامتر به‌صورت آدرس پاراگراف AT به کار می‌رود که آدرس پاراگراف قبلاً باید تعریف‌شده باشد. در این حالت می‌توان برچسب‌ها و متغیرها را در آفست ثابتی از ناحیه حافظه تعریف کرد. به‌عنوان‌مثال، ناحیه حافظه نمایش به‌صورت VIDEO-RAM SEGMENT AT O B800h تعریف می‌شود.

---

[1] align
[2] combine
[3] class

۴-۵-۳ پارامتر کلاس

پارامترهای کلاس به پیونددهنده کمک می‌کنند تا سگمنت هایی بانام‌های مختلف را به هم پیوند دهد، سگمنت ها را مشخص کند و ترتیب آن‌ها را کنترل کند. کلاس می‌تواند نام معتبری داشته باشد که در نقل‌قول قرار می‌گیرد، مثل 'CODE' و 'DATA' اگر پارامتر کلاس برابر 'CODE' باشد، پیونددهنده انتظار دارد که این سگمنت حاوی دستورات اسمبلی باشد. دستورات زیر را در نظر بگیرید:

```
Datasg segment para public 'data'
…  …
Datasg ends

Stacksg segment para stack 'stack'
…  …
Stacksg ends
```

در دستور اول، سگمنت داده‌ای تعریف می‌شود که آدرس شروع آن بر ۱۶ قابل قسمت است (para) و هنگام لینک شدن با برنامه دیگر، ناحیه داده آن برنامه، به‌طور متوالی در حافظه قرار می‌گیرد. دستور دوم سگمنت پشته‌ای را تعریف می‌کند که آدرس شروع آن نیز بر ۱۶ قابل قسمت است.

۴-۶ ویژگی‌های سگمنت کُد و تعریف روال[1]

سگمنت کد حاوی دستورالعمل‌های برنامه است که به زبان ماشین ترجمه و اجرا می‌شود سگمنت کد حاوی یک یا چند روال است. به‌طورکلی، هر برنامه اسمبلی حداقل از یک‌روال تشکیل‌شده است و باید با دستور proc تعریف گردد. دستور proc به‌صورت زیر به کار می‌رود:

```
نام روال      proc    far
    …
نام روال      endp
```

نام روال یک شناسه است و مانند شناسه نام‌گذاری می‌شود. تنها روال برنامه با دستور proc و با پارامتر far شروع می‌شود و به endp خاتمه می‌یابد. فعلاً توجه به مفهوم far چندان مهم نیست ولی بدانید که در تعریف تنها روال برنامه، باید از far استفاده نمایید. بنابراین، سگمنت کدی که از یک‌روال تشکیل‌شده باشد به‌صورت زیر تعریف می‌شود:

```
Codesg segment para none 'code'
Pname proc far
    …
Pname endp
Codesg ends
```

[1] Procedure

۴-۷ تعیین اهداف هر سگمنت

پس از تعیین سگمنت های موردنیاز برنامه، باید هدف آن‌ها نیز مشخص شود. به‌عبارت‌دیگر، باید هر سگمنت را به ثبات سگمنت مربوط کنیم. یعنی سگمنت کد را به ثبات سگمنت کد، سگمنت داده را به ثبات سگمنت داده، سگمنت اضافی را به ثبات سگمنت اضافی و درنهایت سگمنت پشته را به ثبات سگمنت پشته مربوط کنیم. برای این منظور از دستور ASSUME. به‌صورت زیر استفاده می‌شود:

Assume cs: سگمنت کد, ds: سگمنت داده, es: سگمنت اضافی, ss: سگمنت پشته

- توجه داشته باشید که اگر برنامه شما فاقد سگمنت خاصی بود، لازم نیست آن را به ثبات مربوطه‌اش مرتبط کنید. به‌عنوان‌مثال اگر برنامه‌ای فاقد سگمنت پشته بود، لازم نیست ثبات ss در دستور assume ظاهر شود.
- دستور assume در بخش سگمنت کد قرار می‌گیرد.

نمونه‌ای از کاربرد آن را در زیر مشاهده می‌کنید:

```
Stacksg segment para stack 'stack'
    …
Stacksg ends

Datasg segment para 'data'
    …
Datasg ends

Codesg segment para 'code'
  Proc1 proc far
    Assume cs: codesg, ds: datasg, ss: stacksg
    …
  Proc1 endp
Codesg ends
    End proc1
```

وقتی دستور assume سگمنت ها را با ثبات‌های سگمنت مربوط کرد، اسمبلر می‌تواند آدرس‌های آفست عناصر موجود در سگمنت کد، سگمنت داده‌ها و سگمنت پشته را تعیین کند. به‌عنوان‌مثال وقتی stacksg به ثبات ss نسبت داده می‌شود، پردازنده از آدرس موجود در ss برای تعیین آدرس پشته استفاده می‌کند.

۴-۸ تعریف متغیرها در سگمنت داده

گاهی ممکن است لازم باشد در برنامه اسمبلی، داده‌هایی را در برنامه تعریف کنید. برای این کار باید آن‌ها را در سگمنت داده‌ها قرار دهید. این مقادیر ممکن است رشته‌ای یا عددی باشند. ثوابت عددی ممکن است در مبنای ۲، مبنای ۸، مبنای ۱۰ یا مبنای ۱۶ باشند. در اسمبلی ثوابت به‌صورت مبنای ۱۰ فرض می‌شوند، مگر اینکه پسوند آن‌ها را با حروف h، b، q یا o مشخص کنید. h به معنای ۱۶، b به معنای ۲ و q و o به مبنای ۸ هستند. این پسوندها می‌توانند به‌صورت حروف کوچک یا بزرگ نوشته شوند.

۴-۹ تعریف داده‌ها با دستور DB

دستور DB برای تعریف داده‌های یک بایتی به کار می‌رود:

شناسه DB مقدار

شناسه، نام ثابت است، به‌عبارت‌دیگر شناسه، محلی از حافظه در سگمنت داده‌هاست که مقدار موردنظر در آنجا ذخیره می‌شود. مقادیر عددی که می‌توانند توسط DB تعریف شوند، در بازه ۲۵۵- تا ۲۵۵ (در مبنای ۱۰) هستند. اگر عدد فاقد علامت باشد، از صفرتا ۲۵۵ قابل‌استفاده است. اعداد منفی ۱۲۸- تا ۱- نیز در یک باید ذخیره می‌شوند، اما به اعداد منفی ۲۵۵- تا ۱۲۹- یک کلمه اختصاص می‌یابد ولی بایت باارزش آن نادیده گرفته خواهد شد. دستورات زیر را در نظر بگیرید:

```
Place1    db    0          ;    value is 00
Palce2    db    -128       ;    value is 80
Place3    db    255        ;    value is FF
Place4    db    +91        ;    value is A5
Palce5    db    01111101B  ;    value is 7D
```

در دستور DB می‌توان مقادیر کاراکتری (یک کاراکتر) و رشته‌ای (چند کاراکتر) را تعریف کرد. برای تعریف کاراکترها یا رشته‌ها از علامت نقل‌قول یکانی (') یا علامت نقل دوتایی (") استفاده می‌شود. خود این کاراکترها نیز قابل‌استفاده‌اند. اگر رشته‌ای به کوتیشن دوتایی محصور می‌شود، وجود کوتیشن یکانی در آن، به‌عنوان یک کاراکتر داده محسوب، مثل "ali's book" و اگر رشته‌ای به کوتیشن یکانی محدود می‌شود، کوتیشن دوتایی موجود در آن به‌عنوان یک‌قلم داده محسوب خواهد شد. مثل 'ali"s book'. به مثال‌هایی از تعریف رشته توسط DB توجه کنید:

```
Str1  DB    'x'
Str2  DB    "ali's book"
Str3  DB    "my first sample."
```

تعریف چند مقدار در دستور DB امکان‌پذیر است. دستور زیر را ببینید:

```
P1    DB    5, 15, 30
```

این دستور سه بایت از حافظه را تعریف کرده مقادیر 05h، 0Fh و 1Eh را در آن‌ها قرار می‌دهد.

۴-۱۰ تعریف داده‌ها با دستور DW

دستور DW برای تعریف داده‌های یک‌کلمه‌ای به کار می‌رود:

شناسه DW مقدار

بازه قابل‌قبول عددی برای DW، از ۶۵۵۳۵- تا ۶۵۵۳۵ است. اگر اعداد بدون علامت باشند، بازه آن از صفرتا ۶۵۵۳۵ است. به اعداد منفی ۳۲۷۶۸- تا ۱- یک کلمه اختصاص می‌یابد. به اعداد منفی ۶۵۵۳۵- تا ۳۲۷۶۹- چهار بایت اختصاص می‌یابد ولی از دو بایت بالایی صرف‌نظر می‌شود. دستورات زیر را در نظر بگیرید:

```
Word1     DW    -32768 ;    value is 8000
Word2     DW     65535 ;    value is FFFF
Word3     DW    -10000 ;    value is FC18
Word4     DW    -40000 ;    value is 63C0
```

دقت داشته باشید که عدد ۴۰۰۰۰- ، در حافظه چهار بایت را به‌صورت زیر اشغال می‌کند و فقط دو بایت سمت راست مورداستفاده قرار می‌گیرد:

FF FF 63C0

اگر برای تعریف کاراکتر از دستور DW استفاده شود، کاراکتر موردنظر در بایت کم‌ارزش قرار می‌گیرد و محتویات بایت باارزش برابر 00 خواهد شد. تعریف رشته‌ها نیز با دستور DW امکان‌پذیر است ولی رشته‌هایی که در این دستور تعریف می‌شوند، حداکثر باید دو کاراکتری باشند. مثل دستورات زیر:

```
Str1 DW   'm'
Str2 DW   "no"
Str3 DW   'ok'
```

تعریف چند مقدار در دستور DW امکان‌پذیر است، به‌عنوان‌مثال، دستور زیر را در نظر بگیرید:

```
PlaceDW   10, 20, 30, 40
```

این دستور، چهار کلمه حافظه را با مقادیر 000Ah، 0014h، 001Eh، 0028h تعریف می‌کند.

اگر در هرکدام از دستورات DB و DW به‌جای مقادیر موردنظر، علامت ؟ به کار گرفته شود، آن محل حافظه به برنامه اختصاص می‌یابد و می‌توان بعداً مقداری را در آن قرارداد (جا برای استفاده بعدی رزرو می‌شود):

```
P1   DW   ?
P2   DB   ?
```

در داخل یک کلمه بایت‌ها به ترتیب معکوس ذخیره می‌شوند. به‌طوری‌که بایت کم‌ارزش در آدرس پایین‌تر قرار دارد. به‌عنوان‌مثال، 1234h در حافظه به‌صورت زیر ذخیره می‌شود:

آفست	**:**	**00**	**01**
مقدار	**:**	**34**	**12**

۴-۱۱ تعریف داده‌ها با دستور DD

این دستور برای تعریف داده‌های ۳۲ بیتی (کلمه مضاعف) به کار می‌رود:

شناسه DD مقدار

مقداری که در این دستور ذکر می‌شود، بین 0 تا 0FFFFFFFh است، دستورات زیر را ببینید:

```
P1   dd   0, 0BCDA1234h,      -2147483648
P2   dd   100h   dup (?)
```

داده‌ها در کلمه مضاعف به ترتیب معکوس قرار می‌گیرند، به‌طوری‌که ارقام کم‌ارزش در آفست پایین‌تر قرار دارند. به‌عنوان‌مثال، مقدار 12345678 به‌صورت زیر ذخیره می‌شود:

آفست	**:**	**00**	**01**	**02**	**03**
مقدار	**:**	**78**	**56**	**34**	**12**

۴-۱۲ عملگر DUP

با استفاده از عملگر DUP در دستورات DB، DW و DD می‌توان چندین بایت یا کلمه از حافظه را مورداستفاده قرارداد. دستورات زیر را در نظر بگیرید:

```
Place1      DW    100     dup (0)
Place2      DB    50      dup ("*")
Place3      DB    25      dup ("* "), '*'
                              ↑
                          فضای خالی
```

دستور اول، ۱۰۰ کلمه از حافظه را اختصاص می‌دهد که محتویات هرکدام از آن‌ها 0000 است. دستور دوم ۵۰ بایت حافظه را اختصاص می‌دهد که محتویات هر بایت برابر با * است. دستور سوم، ۵۱ بایت از حافظه را تعیین می‌کند که حاوی ۲۶ ستاره است و هر ستاره بافاصله از هم جداشده‌اند. اکنون دستور زیر را در نظر بگیرید:

```
P1   db     10     dup (5 dup ('*'), 5 dup ('+'), 5 dup ('-'))
```

این دستور ۵ ستاره، ۵ علامت جمع و سپس علامت منها را چاپ می‌کند و این کار را ۱۰ بار تکرار می‌نماید.

در دستورات DB، DW و DD از عبارات محاسباتی نیز می‌توان استفاده کرد شاید کاربرد آن چندان جالب به نظر نرسد ولی ممکن است گاهی برای خوانایی برنامه خوب باشد. دستورات زیر این عمل را انجام می‌دهند:

```
P1   DW    16
P2   DW    P1 * 80
P3   DW    10 * 2 * 8
```

۴-۱۳ تعریف مقدار برای شناسه

باراهنمای EQU می‌توان مقداری را برای شناسه‌ای تعریف و در برنامه از آن استفاده کرد. راهنمای EQU به‌صورت زیر به کار می‌رود:

شناسه EQU مقدار

به‌عنوان‌مثال، دستور زیر را در نظر بگیرید:

```
Sum         equ    20
```

با اجرای این دستور، مقدار شناسه sum برابر با ۲۰ خواهد بود و در هر جای برنامه که از sum استفاده شود، در زمان ترجمه، اسمبلر آن را با ۲۰ قرار خواهد داد. اکنون دستورات زیر را در نظر بگیرید:

```
TotalDW    125
Sum        equ    total
```

با اجرای دستور اول مقدار ۱۲۵ در total قرار می‌گیرد و دستور دوم موجب می‌شود تا sum و total یکسان باشند، یعنی هر جا که sum به‌عنوان عملوند دستوری ذکر شود، آدرس total منظور خواهد شد.

۴-۱۴ دستور TEXTEQU

برای تعریف مقادیری برای شناسه‌های متنی، از دستور اسمبلر TEXTEQU می‌توان استفاده کرد:

شناسه TEXTEQU متن

دستور زیر، متن 'this is a sample' را به STR اختصاص می‌دهد:

```
STR     textequ 'this is a sample'
```

۴-۱۵ دستور MOV

دستور MOV برای انتقال داده‌ها از محلی به محل دیگر مورداستفاده قرار می‌گیرد و به‌صورت زیر به کار می‌رود:

MOV عملوند اول و عملوند دوم

این دستور محتویات عملوند دوم (منبع) را به عملوند اول (مقصد) منتقل می‌کند. توجه داشته باشید که محتویات قبلی عملوند اول از بین می‌رود ولی محتویات عملوند دوم تغییر نمی‌کند.

در دستور MOV هر دو عملوند به‌طور همزمان نمی‌توانند محل‌هایی از حافظه باشند.

یعنی انتقال می‌تواند از ثبات به حافظه، از حافظه به ثبات، از ثبات به ثبات یا انتقال مقادیر ثابت به ثبات باشد.

به دستورات زیر توجه کنید:

```
Place1      DB    ?
Place2      DW    ?
        MOV     AX , CX          انتقال محتویات ثبات CX به ثبات AX
        MOV     DS , BX          انتقال ثبات BX به ثبات سگمنت داده
        MOV     Place1 , dh      انتقال ثبات به حافظه
        MOV     AX ,40           انتقال یک مقدار به ثبات
        MOV     CH , Place1      انتقال محتویات حافظه به ثبات
        MOV     CX , DS          انتقال ثبات سگمنت به ثبات معمولی
        MOV     Place2 , DS      انتقال ثبات سگمنت به حافظه
```

در این دستورات، شکل‌های مختلف استفاده از دستور MOV مطرح شد. شکل دیگر MOV که با آدرس غیرمستقیم سروکار دارد، در ادامه بحث می‌شود. در استفاده از دستور MOV به نکات زیر توجه کنید:

- طول عملوندهای منبع و مقصد باید یکسان باشد، مثلاً یک بایت از حافظه باید به ثباتی با طول یک بایت منتقل شود و دو بایت از حافظه باید به ثباتی با طول ۲ بایت منتقل شود. یعنی انتقال باید بایت به بایت یا کلمه به کلمه صورت گیرد.
- انتقال از حافظه به حافظه ممکن نیست.
- انتقال ثبات سگمنت به ثبات سگمنت دیگر ممکن نیست.
- انتقال یک مقدار ثابت به ثبات سگمنت امکان‌پذیر نیست.

مثال) برنامه‌ای که دو مقدار ۱۵ و ۲۵ را در محل‌های p1 و p2 قرار می‌دهد و محتویات این دو محل را باهم عوض می‌کند.

توضیح

در این برنامه، نتیجه عمل را نمی‌توانید مشاهده کنید، زیرا خروجی برنامه در صفحه‌نمایش یا چاپگر ظاهر نمی‌شود فعلاً یاد بگیرید که چگونه داده‌ها را در برنامه کامل تعریف کنید، آن‌ها را از حافظه به ثبات ببرید و برعکس، و جای آن‌ها را عوض کنید. در این برنامه دو دستور زیر برای خاتمه دادن به اجرای برنامه و انتقال کنترل به سیستم‌عامل به‌کاررفته‌اند:

```
Mov ax, 4c00h
Int 21h
```

دستور mov مقدار 4c00h را که تابعی از وقفه 21h است در ثبات ax قرار می‌دهد و دستور int وقفه 21h را اجرا می‌کند.

این برنامه را در یک ویراستار متنی، مثل NotePad تایپ کرده و با پسوند asm ذخیره کنید و به روشی که در ادامه می‌آید آن را اجرا کنید و برای اجرای برنامه‌ها باید TASM و TLINK یا MASM و LINK را در اختیار داشته باشید.

توجه کنید که در این برنامه برای تعویض محتویات دو محل از حافظه، از محل‌های کمکی استفاده‌شده است.

```
Stksg   segment                 stack
        DB      32 dup ("stck")
Stksg   ends

Datasg  segment         para    'data'
   P1   DW      15
   P2   DW      25
Datasg ends

Codesg segment          para    'code'

   Mainproc     far
      Assume    ds: datasg,     cs: codesg,     ss: stksg

      Mov       ax, datasg
      Mov       ds, ax

      Mov       ax, p1
      Mov       bx, p2
      Mov       p1, bx
      Mov       p2, ax

      Mov       ax, 4c00h
      Int21h
   Mainendp

Codesg ends
        End     main
```

تعریف سگمنت پشته

تعریف سگمنت داده

قرار دادن آدرس سگمنت داده در ثبات ds

مراحل ترجمه و اجرای برنامه:

1- MASM (ترجمه)
2- LINK (پیوند دادن)
3- (اجرا)

نکته: برای اجرای تمام برنامه‌ها باید این روند را دنبال کنید.

۴-۱۶ دستور LEA

این دستور آدرس آفست متغیری را در ثباتی قرار می‌دهد و کاربرد آن به‌صورت زیر است:

LEA **<ثبات>, <حافظه>**

دستورات زیر را در نظر بگیرید:

```
Table     db    25    dup (?)
Onebyte   db    ?
          Lea   bx, table
          Mov   al, [bx]
          Mov   onebyte, al
```

دستورات اول و دوم محل‌هایی از حافظه را تعریف می‌کنند، دستور سوم آدرس آفست table را در ثبات bx قرار می‌دهد و دستور چهارم محتویات جایی را که bx به آن اشاره می‌کند در al قرار می‌دهد و دستور پنجم مقدار ثبات al را در onebyte قرار می‌دهد. توجه داشته باشید که منظور از آدرس آفست یک متغیر، فاصله آن متغیر از ابتدای سگمنت است. به‌عنوان‌مثال، اگر متغیری مثل x در داخل یک سگمنت در محل 100h قرار داشته باشد، آدرس آفست آن 100h است.

۴-۱۷ عملگر OFFSET

عملگر OFFSET نیز می‌تواند آدرس آفست متغیر را تعیین کند و به تنهای عمل خاصی را انجام نمی‌دهد. این عملگر با دستور MOV، عمل LEA را انجام می‌دهد. به‌عنوان‌مثال، دستور زیر، آدرس آفست table را در ثبات bx قرار می‌دهد.

Mov **bx, offset table**

۴-۱۸ انواع عملوندها

در مثال‌هایی که راجع به دستور MOV مشاهده کردید، بعضی از عملوندها را به‌کاربرده‌اید. به‌طورکلی، عملوندها به سه دسته تقسیم می‌شوند: عملوندهای بلافصل، ثبات و حافظه. عملوند بلافصل، یک مقدار ثابت است. عملوند ثبات، یکی از ثبات‌های CPU است. عملوند حافظه به محلی از حافظه مراجعه می‌کند.

۴-۱۸-۱ عملوندهای بلافصل

اگر عملوند دستوری، مقدار ثابتی باشد، آن عملوند را بلافصل گویند. دستورات زیر را در نظر بگیرید:

```
MOV     AX, 50
MOV     Place2, 30h
```

دستور اول، عدد ۵۰ (مبنای ۱۰) را در ثبات AX و دستور دوم عدد ۳۰ (مبنای ۱۶) را در حافظه‌ای به نام Place2 قرار می‌دهد. عملوندهای بلافصل می‌توانند در مبنای ۲، ۱۰، ۸ یا ۱۶ باشند.

طول عملوند بلافصل نباید از طول عملوند دیگر بزرگ‌تر باشد. به‌عنوان‌مثال دستور زیر نادرست است (زیرا ثابت 0237h دو بایتی است ولی AH یک بایتی است):

```
MOV AH, 0237h
```

۴-۱۸-۲ عملوندهای ثبات

همان‌طور که در دستور mov مشاهده شد، ثبات‌ها می‌توانند به‌عنوان عملوند دستورات مورداستفاده قرار گیرند. در دستورات زیر، ثبات به‌عنوان عملوند منظور شده است:

```
Mov     ax, cx
Mov     dx, place2
Mov     cl, 20h
```

۴-۱۸-۳ عملوندهای حافظه

دستورالعمل‌های اینتل، انواع گوناگونی از عملوندهای حافظه را به کار می‌برند تا با آرایه‌ها و سایر ساختمان داده‌ها به‌خوبی کار کنند. قبل از پرداختن به عملوندهای حافظه به اصطلاحاتی که در آن آمده است توجه کنید:

تفاوت مکان[1] یک عدد یا آفست یک متغیر است. **آدرس مؤثر** یک عملوند، آفست (فاصله) داده‌ها از آغاز سگمنت است. هر نوع عملوند به محتویات حافظه در آدرس مؤثر اشاره می‌کند. Bytelist و list محل‌هایی از حافظه‌اند. **حالت آدرس‌دهی** که توسط دستوری مورداستفاده قرار می‌گیرد، به نوع عملوند حافظه‌ای که به کار می‌رود اشاره می‌کند. به‌عنوان مثلاً دستور زیر، از حالت آدرس‌دهی در ثبات غیرمستقیم استفاده می‌کند:

```
Mov ax, [si]
```

مفهوم این دستور این است که محتویات جایی که ثبات si به آنجا اشاره می‌کند در ثبات ax قرار گیرد.

۴-۱۸-۴ عملوندهای مستقیم حافظه

در دستورات اسمبلی می‌توان از عملوندهایی که مستقیماً به حافظه اشاره می‌کنند استفاده کرد. این نوع عملوندها را **عملوندهای مستقیم حافظه** گویند. به‌عنوان‌مثال دستورات زیر را در نظر بگیرید:

```
Place1      DB      10
Place2      DW      315
            Mov     bx, Place2
            Mov     ah, Place1
            Mov     cx, ds: [3521h]
```

دستور اول، مقدار ۱۰ را در Place1 و دستور دوم مقدار ۳۱۵ را در Place2 قرار می‌دهد. دستور سوم محتویات Place2 را به BX و دستور چهارم محتویات محل Place1 را به AH منتقل می‌کند و دستور پنجم، آفست 3521h

[1] displacement

را با آدرس موجود در ثبات سگمنت داده ds جمع می‌کند تا آدرس محلی از حافظه به دست آید و سپس محتویات آن محل را در ثبات cx قرار می‌دهد.

نوع عملوند	مثال	شرح
مستقیم	Op1 bytelist	آدرس مؤثر، آفست یک متغیر است
مستقیم – آفست	Bytelist + 2	آدرس مؤثر برابر با مجموع آفست متغیر و تفاوت مکان است
غیرمستقیم ثبات	[si] [bx]	آدرس مؤثر، محتویات ثبات پایه یا ثبات اندیس است
اندیس یا پایه	List [bx] [List + bx] List [di] [si + 2]	آدرس مؤثر برابر با مجموع ثبات پایه اندیس و تفاوت مکان است
پایه – اندیس	[bx + di] [bx] [di] [bp – di]	آدرس مؤثر برابر با مجموع ثبات پایه و یک ثبات اندیس است
پایه – اندیس با تفاوت مکان	[bx + si + 2] List [bx + si] List[bx] [si]	آدرس مؤثر برابر با مجموع ثبات پایه، یک ثبات اندیس و یک تفاوت مکان است

۴-۱۸-۵ عملوندهای مستقیم – آفست

در این روش آدرس‌دهی، آفست با یک مقدار (تفاوت مکان) ترکیب می‌شود. یک کاربرد خوب از عملگرهای جمع و تفریق (+ ، -)، دستیابی به لیستی از مقادیر است. عملگر +، مقداری را به آفست یک متغیر اضافه می‌کند. دستورات زیر را در نظر بگیرید:

```
List1 db    0Ah,   0Bh,   0Ch,   0Dh
      Mov   AL,    list1          ; AL = 0Ah
      Mov   BL,    list + 1       ; BL = 0Bh
      Mov   CL,    list + 2       ; CL = 0Ch
```

دستور اول، مقادیری را برای list1 تعریف می‌کند، دستور دوم مقدار 0Ah را در AL، دستور سوم مقدار0Bh را در BL و دستور چهارم مقدار 0CH را در CL قرار می‌دهد.

۴-۱۸-۶ عملوند غیرمستقیم ثبات

عملوند غیرمستقیم، ثباتی است که حاوی آفستِ داده در حافظه است. وقتی آفستِ یک متغیر در ثباتی قرار می‌گیرد آن ثبات به‌عنوان اشاره‌گر به آن عمل می‌کند. این نوع آدرس‌دهی، معمولاً برای دستیابی به متغیرهایی که از چندین مقدار تشکیل‌شده‌اند، مثل آرایه، قابل‌استفاده است. ثبات‌های SI، DI، BX وBP می‌توانند به‌عنوان عملوند غیرمستقیم مورداستفاده قرار گیرند. اگر پردازنده‌های ۳۸۶ و به بعد در اختیاردارید، می‌توانید از ثبات‌های عمومی ۳۲ بیتی نیز در آدرس‌دهی غیرمستقیم استفاده کنید. برای دستیابی به آدرس آفست یک متغیر، از عملگر OFFSET استفاده می‌شود. به‌عنوان‌مثال دستورات زیر را در نظر بگیرید:

```
Var1 DW    50
     Mov   bx, offset var1
     Mov   ax, [bx]
```

با اجرای دستور اول، متغیر var1 با مقدار ۵۰ تعریف می‌شود و با اجرای دستور دوم، آدرس آفست متغیر var1 در ثبات bx قرار می‌گیرد. با اجرای دستور سوم، محتویات محلی از حافظه آدرس آن در bx قرار دارد، در ثبات ax قرار می‌گیرد. چون آدرس متغیر var در ثبات bx است و محتویات آن نیز ۵۰ است، لذا مقدار ۵۰ در ثبات ax قرار داده می‌شود.

آفست‌هایی که توسط عملوندهای غیرمستقیم ایجاد می‌شوند، از ثبات DS منظور خواهند شد، مگر اینکه BP به‌عنوان بخشی از عملوند غیرمستقیم منظور شود. در مجموعه دستورات زیر، با فرض این‌که سگمنت پشته و سگمنت داده در محل‌های مختلفی باشند، دستورات دوم و سوم به محل‌های متفاوتی مراجعه می‌کنند:

```
Mov si, bp          ; si and bp
Mov dl, [si]        ; looks in the data segment
Mov dl, [bp]        ; looks in the stack segment
```

همان‌طور که در این دستورات ملاحظه شد ، سگمنت های پیش‌فرضی برای مراجعه وجود دارد. با استفاده از آدرس‌دهی غیرمستقیم می‌توان به داده‌هایی غیر از داده‌های موجود در سگمنت داده‌ها دست‌یافت. دستورات زیر را در نظر بگیرید:

```
Mov al, cs: [si]        ; offsett from cs
Mov ax, es: [edi]       ; offsett from es
Mov bx, fs: [edx]       ; offsett from fs
Mov dl, ss: [di]        ; offsett from ss
Mov al, gs: [ecx]       ; offsett from gs
```

به همین ترتیب، می‌توان از ثبات‌های BP برای دستیابی به داده‌های که DS، CS یا ES به آن‌ها اشاره می‌کند، استفاده کرد. دستورات زیر را در نظر بگیرید:

```
Mov dl, ds: [bp]        ; offset from ds
Mov al, es: [bp]        ; offset from es
Mov dl, cs: [bp]        ; offset from cs
Mov al, fs: [bp]        ; offset from fs
```

۴-۱۸-۷ عملوندهای اندیس یا ثبات پایه

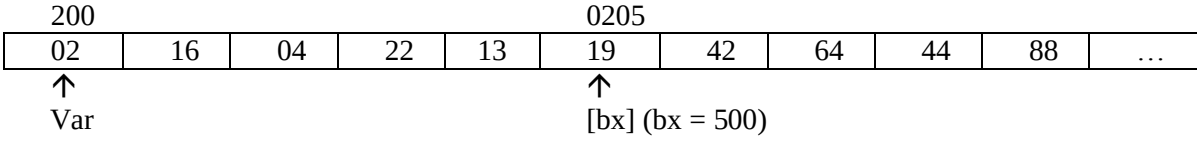
در این روش آدرس‌دهی، ثبات پایه یا ثبات اندیس به یک **تفاوت مکان** اضافه می‌شود. تفاوت مکان، یک مقدار ثابت است که می‌تواند آفست یک متغیر باشد. تفاوت بین پایه و اندیس این است که bx و bp ثبات‌های **پایه** و si و di ثبات‌های **اندیس** هستند. شکل‌های گوناگونی از این روش آدرس‌دهی در اسمبلی قابل‌استفاده است:

ثبات به آفست اضافه شد	ثبات به یک مقدار ثابت اضافه شد
Mov ax, [bx + p1] Mov dx, [bp + 4] Mov dx, 2[si]	Mov dx, var [bx] Mov dx, [di + var] Mov dx, [var + si]

اکنون دستورات زیر را در نظر بگیرید:

```
P1   DW    5
Var  DB    2, 16, 4, 22, 13, 19, 42, 64, 44, 88
     Mov   bx, p1
     Mov al, var [bx]; al=19
```

برای توضیح بیشتر راجع به دستورات فوق، شکل زیر را در نظر بگیرید (در این شکل، آدرس شروع var برابر با 0200 فرض شده است):



در کامپیوترهای ۳۸۶ و به بالاتر، از تمام ثبات‌های عمومی ۳۲ بیتی می‌توانید به‌عنوان ثبات‌های پایه و اندیس استفاده کنید. دستورات زیر را در نظر بگیرید:

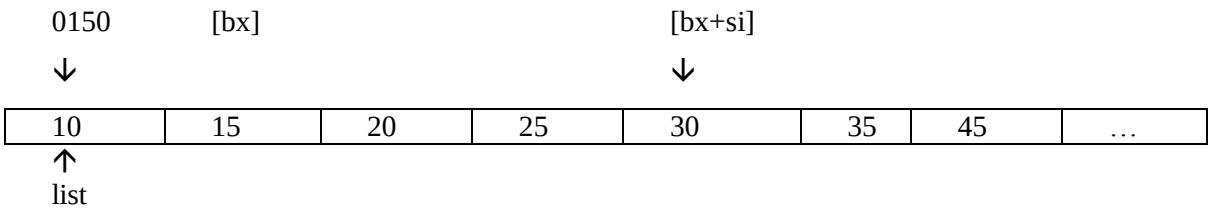
```
Mov ax, [ebx + 3]          ; 16 bit operand
Mov dl, string [edx]  ; 8 bit operand
Mov ecx, [esi]             ; 32 bit operand
Mov eax, mylist [edi]; 32 bit operand
```

۴-۱۸-۸ عملوندهای پایه اندیس

در این نوع آدرس‌دهی، ثبات پایه به ثبات اندیس اضافه می‌شود تا آفست حافظه‌ای به دست آید. این نوع عملوندها برای دستیابی به عناصر آرایه دوبعدی مفید است که در آن ثبات پایه حاوی آفستِ سطر و ثبات اندیس حاوی آفستِ ستون است. دستورات زیر را در نظر بگیرید:

```
List  db      10h, 15h, 20h, 25h, 25h, 30h, 35h, 40h, 45h
Mov bx, offset list          ; points to list
Mov si, 4                    ; set si to 4
Mov al, [bx + si]            ; set the value at bx + si
```

برای آشنایی با عملکرد این دستورات، شکل زیر را در نظر بگیرید (فرض کنید آدرس شروع list، 0150 است):



اولین دستور Mov موجب می‌شود تا bx به ابتدای لیست اشاره کند (0150) دستور Mov بعدی، ثبات اندیس si را برابر ۴ قرار می‌دهد. در آخرین دستور Mov، مقدار موجود در محل bx+si، یعنی محتویات محل 0155 که برابر با ۳۰ است در al قرار می‌گیرد.

توجه به این نکته مهم است که، چون bx و bp ثبات‌های پایه ۱۶ بیتی و si و di ثبات‌های اندیس هستند، یک محدودیت در مورد آن‌ها وجود دارد. آن محدودیت این است که دو ثبات اندیس یا دو ثبات پایه را نمی‌توان باهم ترکیب کرد. بنابراین، دستورات زیر نادرست‌اند:

```
Mov al, [bp + bx]          ; invalid
Mov al, [si + di]          ; invalid
```

۴-۱۸-۹ عملوندهای پایه – اندیس با تفاوت مکان

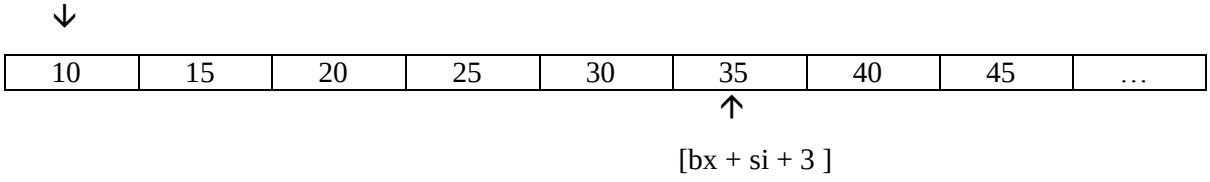
در این روش آدرس‌دهی، ثبات اندیس پایه و تفاوت مکان باهم ترکیب می‌شوند تا آدرس مؤثر به دست آید. شکل‌های متفاوتی از این عملوندها را می‌توان به کاربرد.

```
Mov dx, list[bx] [si]
Mov ax, [bx + si + list]
Mov dl, [bx + si + 3]
Mov cx, list [bp + si]
```

دستورات زیر را در نظر بگیرید:

```
List  db      10h, 15h, 20h, 25h, 30h, 35h, 40h, 45h
      Row DB 5
      Mov bx, offset list
      Mov si, 2
      Mov al, [bx + si + 3]          ; AL=35
```

برای آشنایی با عملکرد این دستورات، شکل زیر را در نظر بگیرید:



۴-۱۹ اعمالی در رابطه با پشته

پشته، حافظه خاصی (در خارج از CPU) است که برای ذخیره موقت داده‌ها و آدرس‌ها به کار می‌رود. پشته در سگمنت پشته قرار می‌گیرد. هر محل ۱۶ بیتی در پشته، توسط ثبات SP (ثبات اشاره‌گر پشته) قابل‌دستیابی است. اشاره‌گر پشته، آدرس آخرین عنصر داده‌ای را که باید در پشته قرار گیرد نگهداری می‌کند. قرار دادن عنصری را در پشته، عمل PUSH گویند. آخرین مقداری که در پشته قرار گرفت، اولین مقداری است که از پشته حذف می‌شود. حذف عنصری را از پشته عمل POP گویند.

چون آخرین مقداری که در پشته قرار گرفت، اولین مقداری است که از پشته حذف می‌شود، می‌گوییم در پشته قانون خروج به ترتیب ورود حاکم است و آن را ساختمان داده LIFO[1] گویند.

پشته کاربردهای جانبی دارد، ازجمله:

- محتویات ثبات‌ها را به‌طور موقت ذخیره می‌کند، به‌طوری‌که می‌توانید از ثبات‌ها برای ذخیره و بازیابی داده‌ها استفاده کنید.

[1] Last In First Out

- وقتی زیر برنامه‌ای فراخوانی می‌شود. CPU آدرس برگشت زیر برنامه به برنامه را در پشته ذخیره می‌کند تا پس از اتمام اجرای زیر برنامه، اجرای برنامه از آن نقطه از سر گرفته شود.
- برای ارسال پارامترها به زیر برنامه به کار می‌رود. به‌طوری‌که می‌توانید پارامترها را داخل برنامه در پشته قرار دهید و در داخل زیر برنامه از پشته بردارید. این کار در زبان‌های سطح بالا نیز صورت می‌گیرد.

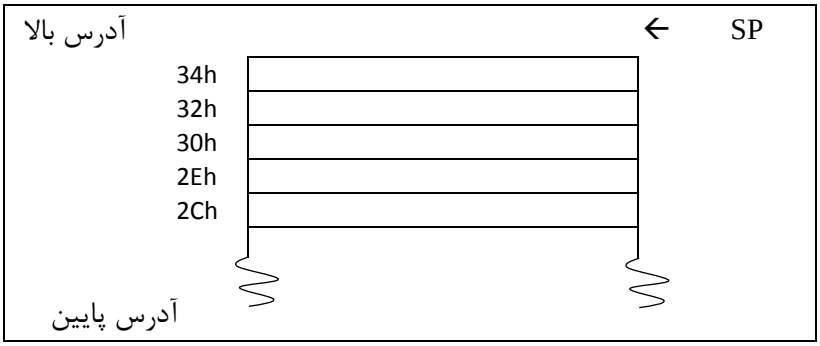
۴-۱۹-۱ دستور PUSH

برای انجام عمل PUSH در پشته (قرار دادن عنصری در پشته)، از دستور PUSH استفاده می‌شود که عملوند آن در پشته قرار می‌گیرد.:

PUSH **<عملوند>**

برای اینکه با عملکرد دستور PUSH آشنا شوید، به نکات زیر توجه کنید:

- ثبات SS آدرس شروع سگمنت پشته را نگهداری می‌کند.
- در آغاز کار، SP حاوی اندازه پشته است و به قبل از انتهای پشته اشاره می‌کند. نحوه ذخیره داده‌ها در سگمنت پشته، متفاوت از سایر سگمنت ها است، به‌طوری‌که داده‌ها از بالاترین آدرس به پایین‌ترین آدرس ذخیره می‌شوند.
- بخشی از پشته که به رویه‌ای اختصاص می‌یابد تا داده‌های آن رویه در آن ذخیره شوند، **قاب پشته**[1] نامیده می‌شود. شکل زیر تصویر یک پشته خالی را نشان می‌دهد که SP به قبل از قاب پشته اشاره می‌کند. فرض می‌کنیم قالب پشته در آفست 34h قرار دارد:

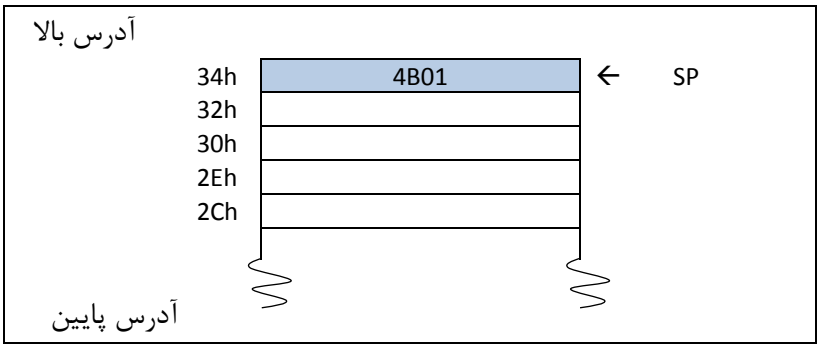


دستورات زیر را در نظر بگیرید:

```
Mov ax, 014Bh
Mov bx, 03C2h
Push ax
```

[1] Stack frame

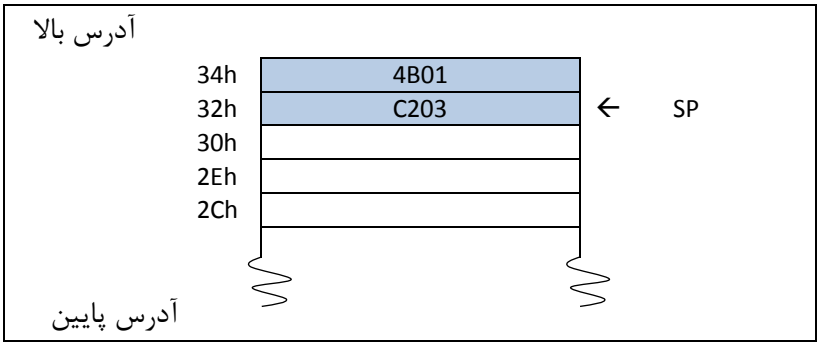
دستور اول مقدار 014Bh را در ثبات ax و دستور دوم مقدار 03C2h را در ثبات bx قرار می‌دهد. دستور سوم مقدار ثبات ax را در پشته قرار می‌دهد و پشته به‌صورت زیر درمی‌آید (داده‌ها به ترتیب معکوس بایت ذخیره می‌شوند):



اکنون دستور زیر را در نظر بگیرید:

```
Push bx
```

این دستور موجب می‌شود محتویات ثبات bx در پشته قرار گیرد و پشته به‌صورت زیر درآید.



همان‌طور که در دو دستور PUSH دیدید، قبل از این‌که مقداری در پشته قرار گیرد، به ثبات SP اضافه می‌شود (مقداری که به ثبات SP اضافه می‌شود، به طول داده بستگی دارد).

۴-۱۹-۲ دستور POP

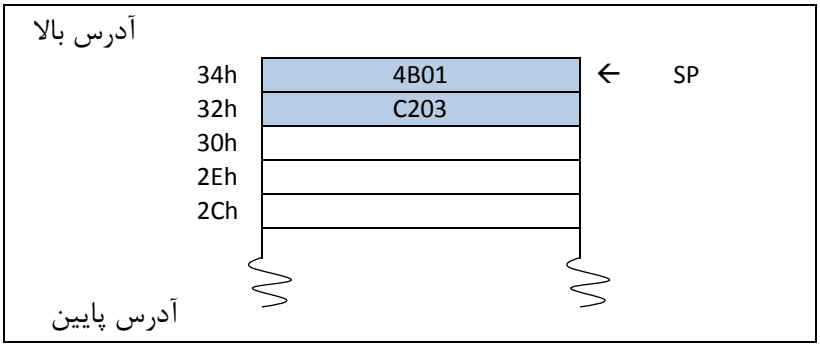
برای حذف عنصری از پشته، از دستور POP به‌صورت زیر استفاده می‌شود:

POP <عملوند>

دستور زیر را در نظر بگیرید:

```
POP          ax
```

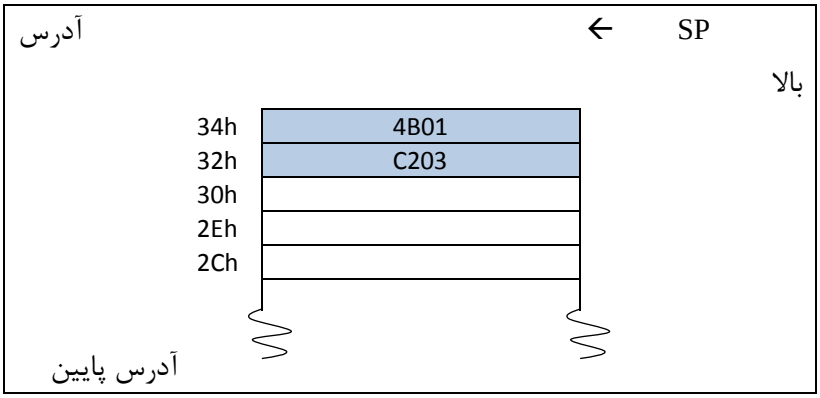
این دستور موجب می‌شود مقداری که SP به آن اشاره می‌کند، در ثبات ax قرار گیرد. با توجه به محتویات قبلی پشته، این مقدار برابر با 03C2 است (بایت‌ها به‌درستی بازیابی می‌شوند). اکنون از SP به‌اندازه ۲ واحد کم می‌شود (مقداری که از SP کم می‌شود، به طول داده بستگی دارد):



اکنون دستور زیر را اجرا می‌کنیم:

```
POP bx
```

با اجرای این دستور، محتویات جایی که SP به آن اشاره می‌کند، در ثبات bx قرار می‌گیرد که برابر با 014B است (کلمه به‌درستی بازیابی می‌شود) و از مقدار SP دو واحد کم می‌شود:



اکنون پشته خالی است، زیرا SP به هیچ‌کدام از مقادیر موجود در پشته اشاره نمی‌کند. همان‌طور که می‌بینید، با علم POP مقادیر موجود در پشته به‌طور فیزیکی حذف نمی‌شوند، بلکه وجود دارند ولی قابل‌دسترسی نیستند.

۴-۱۹-۳ دستورات PUSHF و POPF

دستور PUSHF ثبات فلگ را در پشته قرار می‌دهد و دستور POPF ثبات فلگ را که در پشته ذخیره‌شده است بازیابی می‌کند. این دستورات به‌صورت زیر به کار می‌روند:

```
PUSHF
POPF
```

این دستورات معمولاً در فراخوانی زیر برنامه‌ها مورداستفاده قرار می‌گیرند.

۴-۱۹-۴ دستورات PUSHA و POPA

دستور PUSHA که در پردازنده‌های ۸۰۲۸۶ معرفی شد، ثبات‌های AX، CX، DX، BX، SP، BP، SI و DI را به ترتیب در پشته قرار می‌دهد و ۱۶ واحد به SP می‌افزاید. دستور POPA همین ثبات‌ها را به ترتیب معکوس بازیابی می‌کند و ۱۶ واحد از SP می‌کاهد. نحوه کاربرد این دستورات به‌صورت زیر است:

```
PUSHA
POPA
```

۴-۲۰-۴ دستورات INC و DEC

با استفاده از دستورات INC و DEC می‌توان محتویات ثبات یا محلی از حافظه را یک واحد افزایش یا کاهش داد این دستورات به‌صورت زیر به کار می‌روند:

INC عملوند

DEC عملوند

در هر یک از این دو دستور، عملوند می‌تواند ثبات یا محلی از حافظه باشد. به‌عنوان‌مثال، دستورات زیر را در نظر بگیرید:

```
Label1      DW    10
Label2      DB    15
      Mov ax, Label1
      Inc ax
      Dec Label2
```

دستور اول مقدار ۱۰ را در label1 و دستور دوم مقدار ۱۵ را در label2 قرار می‌دهد. دستور سوم مقدار موجود در label1 را در ax قرار می‌دهد و دستور چهارم به محتویات ax (۱۰) یک واحد اضافه می‌کند. دستور پنجم از label2 یک واحد کم می‌کند و مقدار آن را به ۱۴ تبدیل می‌کند.

مثال) برنامه‌ای که دو مقدار ۲۰ و ۲۵ را در محل‌های حافظه field1 و field2 قرار می‌دهد و سپس به محتویات هر محل یک واحد اضافه می‌کند.

```
Stksg       segment       stack
      Db      32      dup ("stck")
Stksg ends
Datasg      segment       para     'data'
      Field1  DW    20
      Field2  DW    25
Datasg      ends
Codesg      segment para  'code'
   Main     proc    far
      Assume ds: datasg,      cs: codesg,      ss: stksg
      Inc field1
         Inc field2
         Mov ax, 4c00h
         Int 21h
   Main endp
Codesg      ends
      End main
```

مثال) برنامه‌ای که مقادیر ۱۰، ۲۰، ۳۰ و ۴۰ را در حافظه کامپیوتر قرار می‌دهد، سپس آن‌ها را به ترتیب به ثبات‌های AL، AH، BL، BH منتقل کرده، محتویات ثبات‌های AH را با AL و BH را با BL عوض می‌کند. سپس یک واحد به ثبات‌های AH و BH اضافه می‌کند و یک واحد از ثبات‌های AL و BL کم می‌کند.

توضیح: هدف از این برنامه آشنایی با تعریف داده‌ها، دستیابی به داده‌ها، انتقال بین ثبات‌ها و حافظه، تعویض محتویات حافظه، افزایش و کاهش ثبات‌هاست. در این برنامه، از متغیر Help به‌عنوان متغیر کمکی، جهت جابجایی محتویات ثبات‌ها استفاده‌شده است.

```
Stksg       segment         stack
      Db    32      dup ("stck")
Stksg       ends

Datasg      segment         para    'data'
      P1    db      10
      P2    db      20
      P3    db      30
      P4    db      40
      Help  db      ?
Datasg ends

   Codesg   segment         para    'code'
   Main     proc    far
      Assume cs: Codesg, Ds: Datasg, ss: stksg
      Mov ax, Datasg
      Mov ds, ax

      Mov al, P1
      Mov ah, P2
      Mov bl, P3
      Mov bh, P4

      Mov help, al
      Mov al, ah
      Mov ah, help

      Mov help, bl
      Mov bl, bh
      Mov bh, help

      Inc ah            ; increment ah
      Inc bh            ; increment bh
      Dec al            ; increment al
      Dec bl            ; increment bl

      Mov Ax, 4c00h
      Int 21h
   Main endp
Codesg Ends
   End Main
```

تعریف سگمنت پشته

تعریف سگمنت داده

قرار دادن آدرس سگمنت داده در ثبات ds

تعویض محتویات al و ah

تعویض محتویات bl و bh

فصل پنجم دستورات ورودی / خروجی (وقفه‌ها)

یکی از جنبه‌های مهم هر زبان، ازجمله زبان اسمبلی که باید موردبررسی قرار گیرد، ورودی و خروجی داده‌هاست. در اسمبلی، ورودی و خروجی داده‌ها با استفاده از وقفه[1] صورت می‌گیرد. لذا پس از تعریف وقفه و چگونگی پیاده‌سازی آن در سیستم، به چگونگی انجام اعمال ورودی و خروجی به‌وسیله وقفه‌ها می‌پردازیم.

۵-۱ مفهوم وقفه و جدول بردار وقفه‌ها

وقفه سیگنالی از دستگاه جانبی یا برنامه در حال اجرا است که عمل خاصی را درخواست می‌کند. وقتی برنامه در حال اجرا وقفه‌ای را دریافت می‌کند، اجرای برنامه به تعویق می‌افتد و محتویات ثبات‌های CS و IP در پشته نگهداری می‌شود و کنترل اجرای برنامه، به زیر برنامه‌ای می‌رود تا به وقفه پاسخ دهد. هر وقفه، زیر برنامه خاصی دارد که به آن پاسخ می‌دهد. پس از اجرای زیر برنامه پاسخگویی به وقفه، محتویات CS و IP از پشته بازیابی می‌شوند و اجرای برنامه‌ای که به تعویق افتاده بود، از سر گرفته می‌شود. وقفه‌ها به‌طورکلی به دودسته‌اند:

- وقفه‌های سیستم
- وقفه‌هایی که برنامه‌نویس می‌تواند آن‌ها را تولید کند

وقفه‌های سیستم به‌نوبه خود به دودسته تقسیم می‌شوند:

۵-۱-۱ وقفه‌های سخت‌افزاری

همان‌طور که از نامشان مشخص است، به‌وسیله سخت‌افزار ایجاد می‌گردد.

۵-۱-۲ وقفه‌های نرم‌افزاری

شامل وقفه‌های DOS و BIOS است، توجه دارید که DOS و BIOS بخش‌هایی از سیستم‌عامل هستند که اعمال خاصی را انجام می‌دهند. بعضی از وقفه‌های BIOS عبارت‌اند از: 5h، 10h، 11h، 13h، 14h، 15h، 16h و 17h. وقفه DOS وقفه شماره 21h است. تعداد ۲۵۶ وقفه در سیستم مورداستفاده قرار می‌گیرد که از صفرتا ۲۵۵ شماره‌گذاری می‌شوند. آدرس زیر برنامه‌هایی که به وقفه پاسخ می‌دهند، در جدولی به نام بردار وقفه[2] قرار دارد. وقتی وقفه‌ای اتفاق می‌افتد، با استفاده از شماره وقفه، آدرس زیر برنامه‌ای که باید به وقفه پاسخ دهد از جدول بردار وقفه پیدا می‌شود و کنترل اجرای برنامه به این آدرس می‌رود تا زیر برنامه وقفه را اجرا کند. چون ۲۵۶ وقفه وجود دارد و آدرس شروع زیر برنامه وقفه، ۳۲ بیتی (۴ بایت) است، میزان حافظه‌ای که به بردار وقفه اختصاص می‌یابد، ۱۰۲۴ = ۴ × ۲۵۶ بایت است و از آدرس 0h تا 3FFh را اشغال می‌کند. یعنی ۱۰۲۴ بایت اولِ حافظه کامپیوتر، به جدول بردار وقفه اختصاص دارد. برای پیدا کردن آدرس شروع زیر برنامه وقفه، در ۴ ضرب می‌شود. عددی که حاصل می‌شود، محلی از جدول بردار وقفه است که آدرس زیر برنامه وقفه در این محل وجود دارد. به‌عنوان‌مثال، آدرس زیر برنامه پاسخگویی به وقفه شماره ۵، در محل 14h قرار دارد، زیرا ۲۰ = ۴ × ۵ و عدد ۲۰ در مبنای ۱۰، برابر با 14h (در مبنای ۱۶) است.

---

[1] Interrupt
[2] Interrupt vector

۵-۱-۳ مفهوم تابع وقفه

هر وقفه ممکن است دارای خدمات مختلفی باشد و بر اساس آن، اعمال گوناگونی را انجام دهد. خدمات مختلف هر وقفه را تابع آن وقفه می‌گویند و تابع وقفه‌ها نیز از صفر شماره‌گذاری می‌شوند. مثلاً وقفه شماره 10h دارای توابعی با شماره‌های صفر، ۱، ۲، ۳ و غیره است. وقفه 21h نیز توابع متعددی دارد.

۵-۱-۴ اجرای وقفه‌ها در زبان اسمبلی

برای اجرای وقفه‌ها در زبان اسمبلی از دستور INT به‌صورت زیر استفاده می‌شود:

INT شماره وقفه

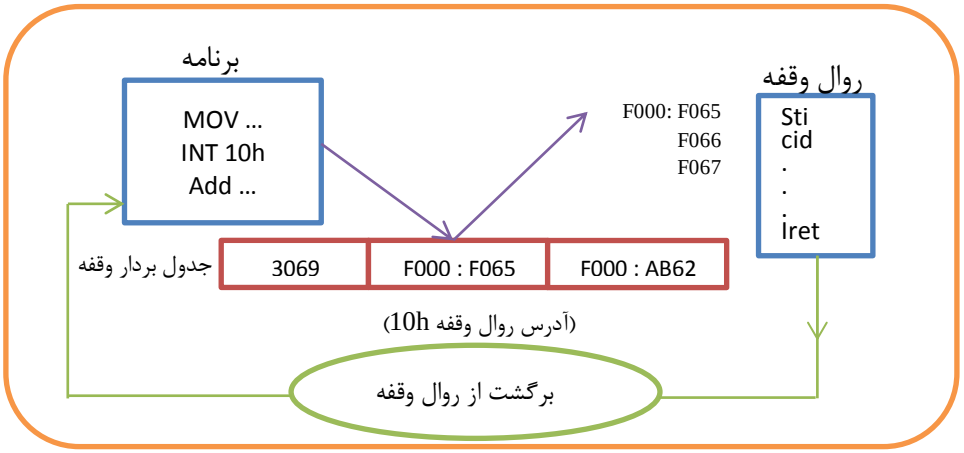
اگر وقفه‌ای که توسط دستور INT اجرا می‌شود، دارای تابع باشد، برای اجرای تابعی از وقفه، شماره تابع باید در ثبات AH قرار گیرد. به‌عنوان‌مثال، برای اجرای تابع 4Ch وقفه 21h باید از دستورات زیر استفاده کرد (این تابع به اجرای برنامه خاتمه می‌دهد):

```
Mov ax, 4C00h
Int 21h
```

۵-۱-۵ مراحل اجرای وقفه

اکنون می‌خواهیم ببینیم که پس از اجرای دستور INT چه اتفاقی می‌افتد. پس از اجرای وقفه، آدرس روال وقفه از بردار وقفه پیدا می‌شود و پس از اجرای روال وقفه، کنترل اجرای برنامه، به دستور بعد از INT برمی‌گردد. می‌توان مراحل اجرای وقفه را به‌صورت زیر بیان کرد:

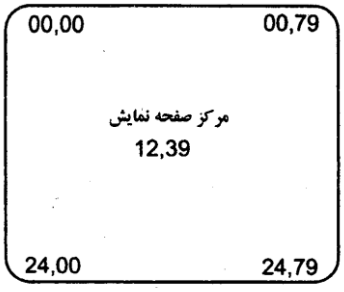
- شماره وقفه که در دستور INT ذکر می‌شود به CPU می‌گوید که به کدام محل از جدول بردار وقفه مراجعه کند. (IP و CS ذخیره می‌شود)
- CPU به آدرسی که در جدول بردار وقفه وجود دارد مراجعه می‌کند. (F000 : F065)
- روال وقفه با شروع از همان آدرس (F000 : F065) اجرا می‌شود تا به دستور IRET برسد.
- دستور IRET موجب می‌شود تا کنترل به دستور بعد از فراخوانی وقفه (برنامه فراخوان) برگردد. (IP و CS بازیابی می‌شود)



چون در این فصل با ورودی و خروجی صفحه‌نمایش سروکار داریم، بررسی مختصری از وضعیت صفحه‌نمایش ضروری به نظر می‌رسد. به همین دلیل ابتدا به تشریح صفحه‌نمایش می‌پردازیم.

۵-۲ صفحه‌نمایش

چون خروجی برنامه باید در صفحه‌نمایش ظاهر شود، باید آشنایی مختصری با آن داشته باشید. این نکته را یادآوری می‌کنیم که، هر صفحه‌نمایش دارای ۲۵ سطر و ۸۰ ستون است. سطرها از صفرتا ۲۴ و ستون‌ها از صفرتا ۷۹ شماره‌گذاری می‌شوند. مختصات صفحه به‌صورت (x ،y) بیان می‌شود که x بیانگر ستون و y بیانگر سطر است. در حافظه سیستم، فضایی به نام فضای نمایش وجود دارد که اطلاعات موجود در آن فضا، در صفحه‌نمایش ظاهر می‌شوند.



۵-۲-۱ تعیین صفات کاراکترها

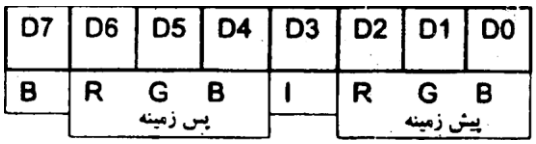
در صفحه‌نمایش رنگی، می‌توان توسط صفت کاراکتر، رنگ زمینه و متن را تعیین کرد. شاید این سؤال مطرح شود که رنگ‌ها چگونه در سیستم تولید می‌شوند؟ رنگ‌های سیستم، از ترکیب سه رنگ اصلی، یعنی قرمز(R)، سبز(G) و آبی(B) تولید می‌شوند. اگر شدت رنگ(I) نیز به این‌ها اضافه شود، از چهار بیت حاصل، ۱۶ رنگ تولید می‌شود. شیوه ترکیب این رنگ‌ها و شدت نور، در جدول زیر آمده است:

شماره رنگ	رنگ	آبی	سبز	قرمز	شدت
۰	سیاه	۰	۰	۰	۰
۱	آبی	۱	۰	۰	۰
۲	سبز	۰	۱	۰	۰
۳	کبود	۱	۱	۰	۰
۴	قرمز	۰	۰	۱	۰
۵	بنفش	۱	۰	۱	۰
۶	قهوه‌ای	۰	۱	۱	۰
۷	سفید	۱	۱	۱	۰
۸	خاکستری	۰	۰	۰	۱
۹	آبی روشن	۱	۰	۰	۱
۱۰	سبز روشن	۰	۱	۰	۱
۱۱	کبود روشن	۱	۱	۰	۱
۱۲	قرمز روشن	۰	۰	۱	۱
۱۳	بنفش روشن	۱	۰	۱	۱
۱۴	زرد	۰	۱	۱	۱
۱۵	سفید شدت بالا	۱	۱	۱	۱

در صفحه‌نمایش رنگی که از رنگ‌ها نیز استفاده می‌شود، شدت نور و چشمک‌زن، در رنگ مؤثر است. زمینه می‌تواند یکی از رنگ‌های صفرتا ۷ باشد. ولی رنگ متن می‌تواند هر یک از ۱۶ رنگ را به خود اختصاص دهد.

توجه به این نکته مهم است که، وقتی صفتی را برای بایتی تعریف می‌کنید تا زمانی که آن را تغییر نداده باشید. به همان حال باقی می‌ماند. نمونه‌هایی از کاربرد رنگ‌ها را در ورودی خروجی اطلاعات خواهید دید.

متن					زمینه					مبنای ۱۶
	I	R	G	B		I	R	G	B	
(سیاه)	0	0	0	0	(سیاه)	0	0	0	0	00H
(آبی)	0	0	0	1	(سفید)	0	1	1	1	71H
(خاکستری)	1	0	0	0	(سبز)	1	0	1	0	A8H
(قرمز)	0	1	0	0	(سفید شدت بالا)	1	1	1	1	F4H
(قرمز)	0	1	0	0	(آبی)	0	0	0	1	14H



۵-۲-۲ پاک کردن صفحه‌نمایش

در ورود و خروج داده‌ها، پاک کردن صفحه‌نمایش از اهمیت خاصی برخوردار است. در این بخش وقفه‌ای را که موجب پاک کردن صفحه‌نمایش می‌شود موردبررسی قرار داده، برنامه‌هایی را می‌نویسیم. تابع 06h وقفه 10h برای پاک کردن صفحه‌نمایش مورداستفاده قرار می‌گیرد. برای استفاده از این تابع، ثبات‌ها باید مانند جدول زیر مقدار بگیرند:

وقفه: 10h	تابع: 6h	عمل تابع: پاک کردن صفحه‌نمایش
6H: AH		
AL: تعداد خطوطی که باید پاک شود		
CH: شماره سطر گوشه بالای سمت چپ		
CL: شماره ستون گوشه بالای سمت چپ		
DH: شماره سطر گوشه پایین سمت راست		
DL: شماره ستون گوشه پایین سمت راست		
BH: صفت محدوده‌ای که باید پاک شود		

مثال) برنامه‌ای که صفحه‌نمایش را پاک می‌کند. در این برنامه، صفت کاراکترها طوری انتخاب شد که رنگ متن قرمز و رنگ زمینه آبی باشد.

توضیح: برای پاک کردن کل صفحه‌نمایش، ثبات‌ها باید به‌صورت زیر مقدار بگیرند:

ثبات	AH	AL	CH	CL	DH	DL	BH
مقدار	6h	25	0	0	24	79	14h

```
Stksg       segment     stack
      Db    32    dup ("stack")
Stksg       ends

Codesg      segment     para    'code'
  Main      proc    far
    Assume cs: Codesg,      ss: stksg

    Mov     al, 25          ; number of rows
    Mov     ch, 0
    Mov     cl, 0
    Mov     dh, 24          ; row
    Mov     dl, 79          ; column
    Mov     bh, 14h ; attribute
    Mov     ah, 6h          ; clear screen
    Int 10h

    Mov     ax, 4c00h
    Int     21h
  Main      endp
Codesg      ends
      End   main
```

پاک کردن صفحه نمایش بطوری که رنگ زمینه آبی و چنانچه متنی بنویسید قرمز خواهد بود

مثال) برنامه‌ای که بخشی از صفحه‌نمایش را پاک می‌کند.

توضیح: برای پاک کردن محدوده‌ای از صفحه‌نمایش که از نقطه (10، 5) شروع می‌شود و به نقطه (40، 15) خاتمه می‌یابد، ثبات‌ها باید به‌صورت زیر مقدار بگیرند:

ثبات	AH	AL	CH	CL	DH	DL	BH
مقدار	6h	10	5	10	15	40	07

```
Stksg       segment     stack
      Db    32    dup ("stack")
Stksg       ends

Codesg      segment     para    'code'
  Main      proc    far
    Assume cs: codesg,      ss: stksg

    Mov     al, 10          ; number of rows
    Mov     ch, 5
    Mov     cl, 10
    Mov     dh, 15          ; row
    Mov     dl, 40          ; column
    Mov     bh, 07          ; attribute
    Mov     ah, 6h          ; clear screen
    Int 10h
```

پاک کردن محدوده ای از صفات از صفحه نمایش

```
    Mov     ax, 4c00h
    Int     21h
  Main      endp
Codesg      ends
    End     main
```

۵-۲-۳ انتقال مکان‌نما

هنگام اخذ ورودی یا تنظیم خروجی، تعیین محل مکان‌نما در صفحه‌نمایش، ضروری به نظر می‌رسد. به‌این‌ترتیب می‌توان در محل خاصی از صفحه‌نمایش اطلاعات را خواند و یا خروجی را در محل خاصی از صفحه‌نمایش تولید کرد. برای انتقال مکان‌نما در صفحه‌نمایش از تابع 02h وقفه 10h استفاده می‌شود. عملکرد این تابع در جدول زیر آمده است. هنگام اجرای این تابع باید شماره تابع در ثبات ah، شماره سطر در ثبات dh، شماره ستون در ثبات dl و شماره صفحه‌نمایش در ثبات bh قرار گیرد.

وقفه: 10h	تابع: 2h	عمل تابع: انتقال مکان‌نما
2H: AH		
DH: شماره سطری که مکان‌نما باید منتقل شود		
DL: شماره ستونی که مکان‌نما باید منتقل شود		
BH: شماره صفحه‌ای از صفحه‌نمایش		

مثال) برنامه‌ای که مکان‌نما را به سطر و ستون دلخواهی (سطر ۱۰ و ستون ۱۵) منتقل می‌کند.

ثبات	AH	DH	DL	BH
مقدار	2h	10	10	0

```
Stksg       segment         stack
    Db      32      dup ("stack")
Stksg       ends
Codesg      segment         para    'code'
  Main      proc    far
    Assume  cs: codesg,     ss: stksg
    Mov     dh, 15          ; row
    Mov     dl, 40          ; column
    Mov     bh, 0           ; page number
    Mov     ah, 2h          ; cursor move
    Int 10h
    Mov     ax, 4c00h
    Int     21h
  Main      endp
Codesg      ends
    End     main
```

تابع انتقال مکان نما

اجرای وقفه ۱۰ برای انتقال مکان نما

۵-۲-۴ چاپ اطلاعات در صفحه‌نمایش

یکی از کارهای ضروری در هنگام دریافت داده‌ها از ورودی، صادر کردن پیام‌های مناسب است. مثلاً اگر بخواهید نام دانشجویی را از ورودی بخوانید، بهتر است پیامی به‌صورت "Enter the name: " صادر گردد و سپس نام دانشجو در جلوی این پیام درخواست شود. به همین منظور، روش ساده‌ای را برای تولید خروجی مطرح می‌کنیم. برای تولید خروجی، از تابع 9h وقفه 21h استفاده می‌کنیم. این تابع رشته را در صفحه‌نمایش ظاهر می‌کند. رشته‌ای که به این تابع تحویل داده می‌شود باید به '$' ختم شود. یعنی، در این تابع، رشته موردنظر، کاراکتر به کاراکتر به خروجی می‌رود تا به علامت '$' برسد. بنابراین یکی از ضعف‌های این تابع این است که خود علامت '$' را نمی‌توانید در خروجی چاپ کنید. برای چاپ رشته موردنظر، آدرس آن رشته باید در ثبات dx قرار گیرد.

مثال) برنامه‌ای که در سطر شماره ۱۰ و ستون شماره ۳۰، پیام 'This is first output' را چاپ می‌کند.

```
Stksg       segment       stack
      Db    32      dup ("stack")
Stksg       ends

Datasg      segment       para    'data'
      Msg   db      'this is first output', '$'
Datasg      ends

Codesg      segment       para    'code'
  Main      proc    far
      Assume ds: datasg,      cs: codesg,       ss: stksg

      Mov   ax, datasg
      Mov   ds, ax

      Mov   dh, 10          ; row
      Mov   dl, 30          ; column
      Mov   bh, 0           ; page number
      Mov   ah, 2h          ; cursor move
      Int 10h

      Mov   dx, offset Msg
      Mov   ah, 9h
      Int   21h

      Mov   ax, 4c00h
      Int   21h
  Main      endp
Codesg      ends
      End   main
```

انتقال مکان نما

چاپ پیام

تابع 40h از وقفه 21h:

این تابع مشابه تابع 09h برای نوشتن رشته بر روی صفحه‌نمایش استفاده می‌شود ولی بهتر از تابع 09 است. در این تابع مفهوم file handle (دستگیره فایل) استفاده می‌گردد. هر وسیله یا دستگاهی به کمک یک شماره مشخص می‌گردد. که به آن file handle گویند، عدد 0 برای صفحه‌کلید، عدد 1 برای صفحه‌نمایش و عدد 4 برای چاپگر است که این شماره باید در ثبات bx ریخته شود. شماره تابع یعنی عدد 40h طبق معمول در ثبات AH

ریخته می‌شود و آدرس رشته در DX و طول رشته در CX ریخته می‌شود. توجه کنید که در اینجا از دول رشته به‌جای علامت ویژه در انتهای آن استفاده می‌گردد.

مثال) برنامه‌ای بنویسید که به کمک تابع 40h از وقفه 21h پیام !Hello را روی صفحه‌نمایش نشان دهد.

```
Stksg       segment     stack
      DW    64    dup (?)
Stksg       ends
Datasg      segment     para    'data'
      Message       db      "Hello!"
Datasg      ends
Codesg      segment     para    'code'
   Main     proc    far
      Assume ds: datasg,      cs: codesg,      ss: stksg

      Mov   ax, datasg
      Mov   ds, ax

      Mov   bx, 0001h
      Mov   cx, 6
      lea   dx, message
      Mov   ah, 40h
      Int 21h

      Mov   ax, 4c00h
      Int   21h
   Main     endp
Codesg      ends
      End   main
```

چاپ !Hello

۵-۲-۵ ورود و خروج کاراکتر

با استفاده از تابع 01h وقفه 21h می‌توان کاراکتری را از صفحه‌کلید خواند و با استفاده از تابع 02h وقفه 21 می‌توان کاراکتری را در صفحه‌نمایش ظاهر کرد. توجه داشته باشید که تابع 01h ضمن دریافت کاراکتر از صفحه‌کلید، آن را نمایش می‌دهد. اگر بخواهید کاراکتر ورودی، در صفحه‌نمایش ظاهر نشود، از تابع 8h برای خواندن کاراکتر استفاده کنید:

شماره تابع	عملکرد
1h	کاراکتر را از صفحه‌کلید خوانده، در ثبات AL قرار می‌دهد (در صفحه‌نمایش نیز ظاهر می‌کند)
2h	کاراکتری را در صفحه‌نمایش چاپ می‌کند، کاراکتر باید در ثبات DL قرار داشته باشد
8h	کاراکتری را از صفحه‌کلید خوانده، در ثبات AL قرار می‌دهد (در صفحه‌نمایش ظاهر نمی‌کند)
09h	نوشتن رشته در خروجی، آدرس رشته باید در ثبات DX قرار داشته باشد
0Ah	خواندن رشته از صفحه‌کلید
4Ch	این تابع برنامه را خاتمه می‌دهد و کنترل را به سیستم‌عامل برمی‌گرداند
40h	نوشتن رشته در خروجی.

مثال) برنامه‌ای که ضمن صدور پیام، کاراکتری را از صفحه‌کلید خوانده، با پیام دیگری آن کاراکتر را در صفحه‌نمایش ظاهر می‌کند.

توضیح:

در این برنامه از تابع 1h برای خواند کاراکتر و از تابع 2h برای چاپ آن استفاده‌شده است. پس از خواندن کاراکتر آن را از AL به help منتقل کردیم تا از بین نرود، چون از ثبات AL در اجرای وقفه استفاده می‌شود.

```
Stksg       segment         stack
      Db    32      dup ("stack")
Stksg       ends
Datasg      segment         para    'data'
      Msg1  db      'please enter a character: ', '$'
      Msg2  db      'you typed this character: ', '$'
      Help  db      ?
Datasg      ends
Codesg      segment         para    'code'
  Main      proc    far
      Assume ds: datasg,    cs: codesg,     ss: stksg
      Mov   ax, datasg
      Mov   ds, ax
      Mov   dh, 10          ; row
      Mov   dl, 30          ; column
      Mov   bh, 0           ; page number      } انتقال مکان نما
      Mov   ah, 2h          ; cursor move
      Int 10h

      Mov   dx, offset Msg1
      Mov   ah, 9h          } چاپ msg1
      Int   21h

      Mov   ah,     1h
      Int   21h             } خواندن کاراکتر و انتقال آن به help
      mov   help, al

      Mov   dh, 12          ; row
      Mov   dl, 30          ; column
      Mov   bh, 0           ; page number      } انتقال مکان نما
      Mov   ah, 2h          ; cursor move
      Int 10h

      Mov   dx, offset Msg2
      Mov   ah, 9h          } چاپ msg2
      Int   21h

      mov   dl, help
      Mov   ah, 2h          } انتقال کاراکتر از help به dl چاپ کاراکتر
      Int   21h

      Mov   ax, 4c00h
      Int   21h

  Main      endp
Codesg      ends
      End   main
```

۵-۲-۶ روش‌های دیگر ورود و خروج کاراکتر

درروش دیگر برای چاپ کاراکترها در خروجی، استفاده از توابع 09h و 0Ah از وقفه 10h است. تفاوت دو تابع 09h و 0Ah این است که در تابع 09h می‌توان صفتی را برای کاراکتر تعیین کرد ولی در تابع 0Ah از صفت فعلی استفاده می‌شود. شباهت آن‌ها این است که در هر تابع می‌توان کاراکتری را چندین بار در صفحه‌نمایش چاپ کرد. تعداد دفعات چاپ کاراکتر در ثبات CX قرار می‌گیرد. در کاربرد این توابع ثبات‌ها را به‌صورت زیر مقدار دهید:

تابع 09h

ثبات	AH	AL	BL	BH	CX
مقدار	09h	کاراکتر	صفت	شماره صفحه	دفعات چاپ

تابع 0Ah

ثبات	AH	AL	BH	CX
مقدار	0Ah	کاراکتر	شماره صفحه	دفعات چاپ

مثال) برنامه‌ای که با استفاده از تابع 09h از وقفه 10h، کاراکتر مربوط به نماد قلب را تعداد ۲۰ بار در صفحه‌نمایش نشان می‌دهد. نماد قلب با کد اسکی 03h مشخص می‌شود. صفحه‌نمایش نیز پاک می‌شود.

```
Stksg       segment       stack
      Db    32      dup ("stack")
Stksg       ends
Codesg      segment       para     'code'
  Main      proc    far
    Assume cs: codesg,      ss: stksg

    Mov     al, 25          ; number of rows
    Mov     ch, 0
    Mov     cl, 0
    Mov     dh, 24          ; row
    Mov     dl, 79          ; column
    Mov     bh, 07          ; attribute
    Mov     ah, 6h          ; clear screen
    Int 10h                                    } پاک کردن صفحه نمایش

    Mov     ah, 09h ; request display
    Mov     al, 03h         ; heart to be display
    Mov     bh, 0
    Mov     bl, 0f0h        ; attribute
    Mov     cx, 20          ; numver of print
    Int 10h

    Mov     ax, 4c00h
    Int     21h

  Main      endp
Codesg      ends
      End   main
```

۷-۲-۵ خواندن رشته از صفحه‌کلید

هدف از این بخش، پردازش داده‌های رشته‌ای نیست، بلکه هدف این است که با شیوه خواندن یک‌رشته از ورودی آشنا شویم و بتوانیم برنامه‌های ساده‌ای را بنویسیم.

برای خواندن رشته‌ها از صفحه‌کلید، از تابع 0Ah وقفه 21h استفاده می‌شود. برای خواندن رشته‌ای از ورودی، باید لیستی به نام لیست پارامترها تعریف کرد. در لیست پارامترها، حداکثر تعداد کاراکترهای ورودی باید مشخص باشد. سیستم با استفاده از این طول، از پذیرش کاراکترهای اضافی خودداری می‌کند. اگر کاراکترهای اضافی وارد شوند، بوق سیستم را به صدا درمی‌آورد و کاراکترهای اضافی را نمی‌پذیرد. با فشردن کلید Enter ورد داده‌ها خاتمه می‌یابد. تعداد واقعی کاراکترهایی که خوانده شدند، در لیست پارامترها قرار می‌گیرد. عنصر بعدی لیست پارامترها، کاراکترهایی است که از ورودی خوانده‌شده‌اند. بنابراین، لیست پارامترها از عناصر زیر تشکیل‌شده است:

- نام لیست پارامترها که با دستور LABEL تعریف می‌شود. دستور LABEL با پارامتر BYTE موجب می‌شود تا لیست پارامترها بر روی مرز بایت تنظیم شود.
- بایت اول لیست پارامترها حاوی حداکثر تعداد کاراکترهایی است که از ورودی خوانده می‌شود.
- بایت دوم لیست پارامترها تعداد واقعی کاراکترهای خوانده‌شده را نگه می‌دارد.
- بایت سوم، شروع محلی از حافظه است که کاراکترهای تایپ‌شده را نگه می‌دارد.

به‌عنوان‌مثال، لیست پارامتر زیر را در نظر بگیرید:

```
Strlist     label   byte            ; start of parameter list
Max db      20                      ; max characters
Len  db     ?                       ; Actual length
Buffer      db      20      dup (' ') ; memory for characters
```

در لیست پارامتری که تعریف شد، دستور label باعث می‌شود تا آدرس آن در مرز بایت تنظیم شود. نام این لیست، strlist است. شناسه max حداکثر طول لیست، شناسه len تعداد واقعی کاراکترهای خوانده‌شده، و شناسه buffer محلی از حافظه را که کاراکترهای ورودی باید ذخیره شوند مشخص می‌کند.

برای خواندن رشته‌ای از ورودی باید تابع 0Ah را در ثبات AH، و آدرس آفست لیست پارامترها را در ثبات DX قرارداد و سپس وقفه 21h را با دستور int اجرا کرد. با توجه به لیست پارامترهای فوق، خواندن رشته به‌صورت زیر انجام می‌شود:

```
Mov ah, 0ah
Lea  dx, strlist
Int   21h
```

با اجرای این دستورات، برنامه منتظر می‌ماند تا کاراکترهایی از صفحه‌کلید وارد شوند. هر کاراکتری که تایپ می‌شود، در صفحه‌نمایش نیز ظاهر می‌گردد. با فشردن کلید Enter ورود داده‌ها خاتمه می‌یابد. کاراکترهای انتهای خط (که با فشردن Enter ایجاد می‌شود)، در محلی که برای کاراکترهای ورودی در نظر گرفته شد ذخیره می‌شود ولی به‌عنوان تعداد کاراکترهای خوانده‌شده محسوب نخواهد شد. لذا یک بایت از ۲۰ بایتی که برای buffer در نظر گرفته شد، در اختیار این کاراکتر (با کد 0Dh) قرار می‌گیرد. درواقع، می‌توان ۱۹ کاراکتر را از ورودی خواند و در buffer قرارداد.

هنگام ورود کاراکترها، با استفاده از کلید Backspace می‌توان کاراکترهایی را پاک کرد. این کلید، جز کلیدهای خوانده‌شده محسوب نمی‌شود. این تابع، از کلیدهایی مثل F1، F2 و ... HOME و غیره صرف‌نظر می‌کند.

همان‌طور که قبلاً گفته شد، برای چاپ رشته‌ها در صفحه‌نمایش از تابع 09h وقفه 21h استفاده می‌شود.

مثال) برنامه‌ای که صفحه‌نمایش را پاک می‌کند و ضمن صدور پیامی، نام دانشجویی را از ورودی خوانده، نام خوانده‌شده را در محل خاصی چاپ می‌کند.

توضیح: هدف از این برنامه آشنایی با ورودی و خروجی رشته‌ها و ترجیحاً صدور پیام به هنگام خواندن و نوشتن داده‌هاست.

```
Stksg       segment       stack
      Db    32     dup ("stack")
Stksg       ends
Datasg      segment       para    'data'
      Msg1  db     'Enter the student name:',"$"
      Msg2  db     'you entered this name:',"$"
      Strlist label byte                    ; start of parameter list
            Max   db     20
            Len   db     ?
            Buffer db    20      dup (' ')
      Dolar db     '$'
Datasg      ends

Codesg      segment       para    'code'
  Main      proc    far
      Assume ds: datasg,      cs: codesg,       ss: stksg
      Mov   ax, datasg
      Mov   ds, ax

      Mov   al, 25          ; number of rows
      Mov   ch, 0
      Mov   cl, 0
      Mov   dh, 24          ; row
      Mov   dl, 79          ; column
      Mov   bh, 07          ; attribute
      Mov   ah, 6h          ; clear screen
      Int 10h                                   } پاک کردن صفحه نمایش

      Mov   dh, 10          ; row
      Mov   dl, 30          ; column
      Mov   bh, 0           ; page number
      Mov   ah, 2h          ; cursor move
      Int 10h                                   } انتقال مکان نما

      Mov   dx, offset Msg1
      Mov   ah, 9h
      Int   21h                                 } چاپ msg1

      Lea   dx, strlist
      Mov   ah, 0Ah
      Int    21h                                } دریافت نام از ورودی
```

```
        Mov     dh, 12          ; row
        Mov     dl, 30          ; column
        Mov     bh, 0           ; page number
        Mov     ah, 2h          ; cursor move
        Int 10h

        Mov     dx, offset Msg2
        Mov     ah, 9h
        Int     21h

        lea     dx, buffer
        Mov     ah, 9h
        Int     21h

        Mov     ax, 4c00h
        Int     21h
    Main        endp
Codesg          ends
        End     main
```

انتقال مکان نما

چاپ msg2

چاپ نام

۳-۵ نحوه اجرا کردن برنامه‌های زبان اسمبلی

۱- نوشتن برنامه موردنظرتان (به زبان اسمبلی) در Notepad، Wordpad یا ویرایشگر MS-DOS

۲- ذخیره آن در یک فایل (Example.ASM)

۳- اجرای برنامه Command Prompt (cmd) در ویندوز

۴- از طریق Command Prompt (cmd)، برنامه TASM.EXE یا MASM.EXE را اجرا کنید. برای مثال:

```
C:\>TASM Example.ASM
```

۵- اگر در برنامه شما اشکالی وجود داشته باشد، برنامه TASM شمارا از آن آگاه خواهد کرد

۶- پس از برطرف کردن اشکالات احتمالی، خروجی برنامه TASM یا MASM یک فایلی با پسوند OBJ است. سپس برنامه TLINK.EXE یا LINK.EXE را مانند زیر اجرا کنید:

```
C:\>TLINK Example.OBJ
```

۷- سپس فایل Example.exe را اجرا کنید.

نکته:

- به یاد داشته باشید که فایل ASM (برنامه شما) و برنامه TASM(MASM) و TLINK(LINK) در یک دایرکتوری مشابه قرارگرفته باشند.
- شما باید همیشه از Command Prompt (از طریق MS-DOS) برنامه‌های TASM(MASM) یا TLINK(LINK) را اجرا نمایید.
- اگر پس از اجرا کردن TASM(MASM یا TLINK(LINK صفحه Commad prompt بسته شد و به پنجره ویندوز بازگشتید، نگران نباشید! در حال حاضر فایل آبجکت یا exe شما در دایرکتوری ساخته‌شده است. بنابراین شما می‌توانید در Command Prompt فایل آبجکت یا exe خود را اجرا نمایید.

فصل ششم ماکرو (Macro)

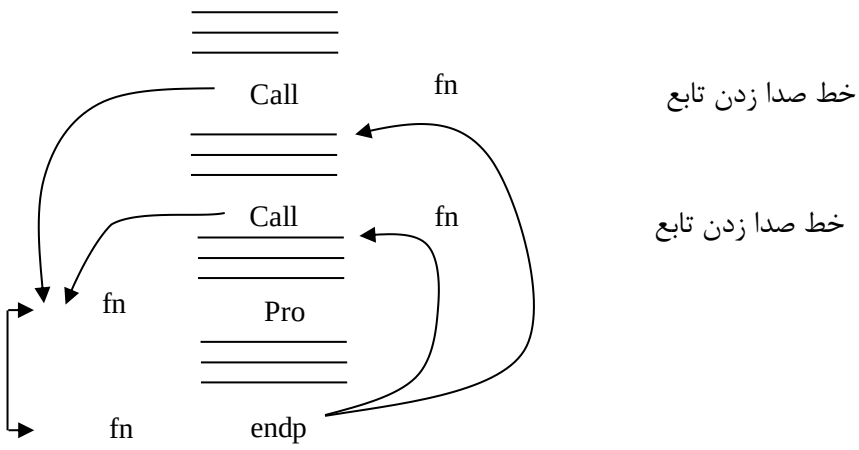
هر دستور زبان اسمبلی که تاکنون معرفی گردید، توسط اسمبلر به یک دستور زبان ماشین ترجمه می‌شود. هر دستور زبان سطح بالا ممکن است توسط کامپایلر به چندین دستور زبان ماشین ترجمه گردد. ماکرو شامل مجموعه‌ای از دستورات است که تعریف می‌شود و هر زمان به این دستورات نیاز باشد به‌راحتی نام ماکرو به‌جای آن‌ها قرارگرفته، اسمبلر دستورات تعریف‌شده درون ماکرو را جایگزین نام ماکرو می‌نماید. بنابراین می‌توان گفت هر دستور زبان سطح بالا یک ماکرو است. مزایای استفاده از ماکرو را در موارد زیر می‌توان خلاصه کرد:

۱- کاهش کدهای تکراری در برنامه
۲- کاهش خطاهای مربوط به استفاده از کدهای تکراری
۳- خواناتر شدن برنامه

۶-۱ تعریف ماکرو

ماکرو نوعی دستورالعمل است که در آن تعدادی از دستورالعمل‌ها، دستورات اسمبلر و یا حتی ماکروهای دیگر قرارگرفته است. اسمبلر ابتدا ماکرو را به دستورات درون آن بسط می‌دهد و سپس این دستورات جدید را اسمبل می‌کند. فرق ماکرو با تابع در این است که کد یک تابع فقط یک‌بار در برنامه نوشته می‌شود و هر بار که آن تابع صدا زده می‌شود یک پرش به آن قسمت صورت می‌گیرد و پس از اتمام کار تابع، کنترل دوباره به محل اولیه برمی‌گردد.

در اینجا تنها کافی است بدانید تابعی به نام fn با دستور Call fn صدا زده‌شده و تعریف آن با دستور proc شروع و با دستور endp خاتمه می‌یابد. پس نحوه عملکرد تابع در زبان اسمبلی به‌صورت کلی زیر است:



ولی هنگامی‌که یک ماکرو صدا زده می‌شود، کد آن ماکرو در محل صدازدن بسط داده می‌شود .
طرز تعریف و صدازدن ماکروها شبیه زبان‌های سطح بالاست.

۶-۲ فرم کلی تعریف ماکرو

ماکرو با شبه دستور MACRO آغازشده و با ENDM پایان می‌یابد. نام ماکرو قبل از شبه دستور MACRO قرار می‌گیرد. این نام بایستی یک نام منحصربه‌فرد در برنامه بود و با قوانین نام‌گذاری زبان اسمبلی مطابقت داشته باشد.

به‌عنوان‌مثال دو دستوری که در ابتدای خیلی از برنامه‌ها همیشه نوشته می‌شوند را می‌توان درون یک ماکرو قرارداد و درون برنامه به‌جای آن دو دستور، فقط نام ماکرو را قرارداد.

```
نام ماکرو      MACRO    پارامترهای ورودی
.
.
.
ENDM
```

مثال)

```
INITZ    MACRO
             Mov    ax, dtseg
             Mov    ds, as
         ENDM
```

داده‌های استفاده‌شده درون ماکرو، باید درجایی دیگر درون برنامه تعریف‌شده باشند، در غیر این صورت باید توسط اسمبلر شناخته‌شده باشند. برای مثال در این ماکرو dtseg بایستی در قسمتی از برنامه تعریف‌شده باشد، بقیه عملوندها نیز که توسط اسمبلر شناخته‌شده هستند، مثلاً ax و ds (چون اسامی رجیسترها هستند). درون برنامه هر جا نیاز به اجرای این دو دستور باشد کافی است نام ماکرو یعنی INITZ نوشته شود. اسمبلر در برخورد با این نام، جدول دستورات نمادین را برای یافتن INITZ جستجو می‌کند و چون آن را نمی‌یابد.، ماکروها را بررسی می‌کند و اگر ماکرو در برنامه تعریف‌شده باشد، دستورات بدنه‌ی ماکرو را جایگزین INITZ می‌نماید. در مثال زیر ماکروی دیگری به نام FINISH نیز تعریف‌شده، مورداستفاده قرارگرفته است.

```
INITZ    MACRO
             Mov    ax, dtseg
             Mov    ds, ax
         ENDM

FINISH   MACRO
             MOV    AX, 4C00h
             Int    21h
         ENDM

STSEG    SEGMENT
    DB   64 DUP (?)
STSEG    ENDS
DTSEG    SEGMENT
    MSG1 DB     'This is an example',"$"
DTSEG    ENDS
CDSEG    SEGMENT
    Main Proc   Far
         AssumeCS: CDSEG,    DS: DTSEG,    SS: STSEG
         INITZ
         MOV    Ah, 09
         Mov    dx, offset MSG1
         Int    21h
         FINISH
    MAIN ENDP
CDSEG    ENDS
    END  MAIN
```

در این مثال مشاهده می‌شود که ماکروها بایستی در ابتدای برنامه تعریف گردند. اگر در فایل لیست تولیدشده توسط اسمبلر برای این برنامه دقت شود، ملاحظه می‌شود که تعاریف ماکرو موقع اسمبل شدن هیچ کد ماشینی تولید نمی‌کنند. بلکه هنگامی‌که ماکرو استفاده‌شده و توسعه پیداکرده است کد ماشین آن تولیدشده است.

۶-۳ پارامترهای ماکرو

گاهی اوقات برای اینکه ماکرو انعطاف بیشتری داشته باشد می‌توانید پارامترهایی را برای ماکروها تعریف کنید. ماکروی زیر، دو آرگومان به نام‌های message (اطلاعاتی که باید نمایش داده شود) و lenght (تعداد کاراکتری که باید نمایش دهد) دارد:

```
Write     macro     message, lenght
          Mov   ah, 40h          ; request display
          Mov   bx, 0001h             ; set device monitor
          Mov   cx, length            ; length
          Lea   dx, message           ; prompt
          Int   21h                   ; call service
Endm
```

اگر تعداد آرگومان‌ها در هنگام تعریف یا فراخوانی ماکرو بیش از یکی باشد، با کاما از هم جدا می‌شوند. ماکروهایی که پارامتر داشته باشند به‌صورت زیر فراخوانی می‌شوند:

نام ماکرو	لیست پارامترهای واقعی

به‌عنوان‌مثال، ماکروی Write به‌صورت زیر فراخوانی می‌گردد:

```
Write     message1, len
```

در هنگام فراخوانی ماکرو، ابتدا نام ماکرو و سپس لیست پارامترهای واقعی آن نوشته می‌شوند و اسمبلر درجایی از ماکرو، که پارامترهای مجازی قرار دارند پارامترهای واقعی را جایگزین می‌نماید. در مثال فوق در هنگام بسط دادن ماکرو اسمبلر به‌جای دستور lea dx , message دستور lea dx , message1 و به‌جای دستور mov cx , lenght دستور mov cx , len را قرار می‌دهد.

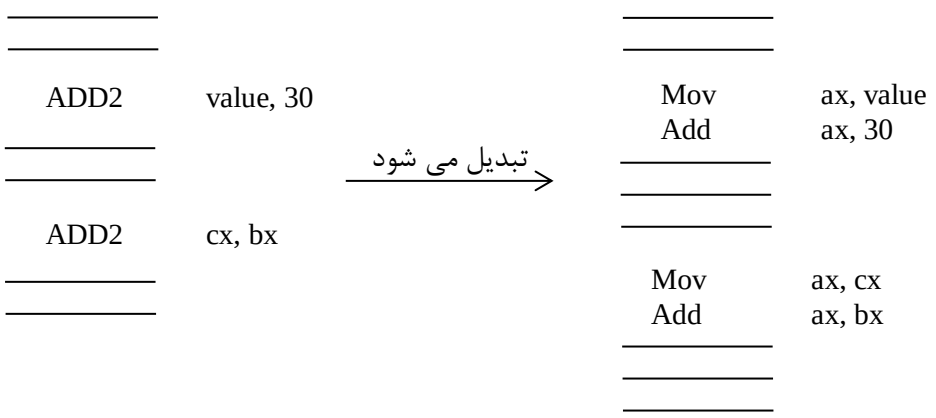
تعداد آرگومان‌های مجاز در هنگام تعریف ماکرو و پارامترهای واقعی آن در هنگام فراخوانی، باید برابر باشد.

بین پارامترهای ورودی علامت کاما (,) قرار می‌گیرد و دور آن‌ها برخلاف زبان‌های سطح بالا علامت پرانتز گذاشته نمی‌شود. تعریف ماکرو باید قبل از صدازدن آن باشد و بهتر است در اول برنامه صورت گیرد.

مثال) ماکرو ADD2 که به‌صورت زیر تعریف‌شده دو پارامتر ورودی را جمع کرده و حاصل را در ثبات ax می‌ریزد:

```
ADD2      MACRO     N1, N2
               Mov   ax,    N1
               ADD   AX,    N2
          ENDM
```

حال اگر در داخل کد سگمنت برنامه ماکروی فوق را ۲ بار صدا بزنیم به‌صورت زیر بسط داده می‌شود:



ازآنجاکه در ماکروها عمل پرش و بازگشت را نداریم، ماکروها سریع‌تر از توابع اجرا می‌شوند، ولی از طرف دیگر استفاده از ماکروها به‌جای توابع طول برنامه را افزایش می‌دهد.

در ماکرو اسمبلر شرکت ماکروسافت، پنج ماکروی مهم و پراستفاده وجود دارد که کار برنامه‌نویسی را سده می‌کنند. معرفی و تعاریف این ماکروها در فایل IO.H و کد ماشین آن‌ها در فایل IO.OBJ قرار دارد، لذا هنگام استفاده از این ماکروها در اول برنامه می‌بایست دستور Include io.h را نوشت و همچنین هنگام لینک برنامه‌ای بانام فرضی test از دستور زیر استفاده کرد:

```
C:\asm> link test.obj + io.obj ↵
```

۴-۶ ماکروی Output

فرم کلی این ماکرو به‌صورت زیر است:

```
OUTPUT    string [, length]
```

این ماکرو برای نمایش رشته‌ای از کاراکترهای مونیتور استفاده می‌شود. String به مکانی در سگمنت داده اشاره می‌کند که حاوی رشته موردنظر است. عملوند اختیاری length تعداد کاراکترهایی که باید نمایش داده شوند را تعیین می‌کند و اگر حذف شود آنگاه رشته باید به کاراکتر NULL (00) ختم شده باشد (مانند زبان C). وجود یک کاراکتر NULL به عمل نمایش رشته پایان می‌دهد حتی اگر عملوند length مورداستفاده قرارگرفته باشد. Length می‌تواند یک عدد صریح، ثبات یا متغیر 16 بیتی باشد.

تذکر:

- این ماکرو محتوای هیچ ثباتی و همچنین ثبات Flag را تغییر نمی‌دهد.

مثال)

```
Include io.h
Sseg segment stack
    Db    256 Dup (?)
Sseg ends
Dseg segment
Message db 'this is a test', 0DH, 0AH, 0
Dseg ends
```

```
Cseg segment
        Assume cs: cseg, ds: dseg
Start:  mov    ax, seg dseg
        Mov    ds, ax
        Output message
        Mov    ax, 4c00h
        Int    21h
Cseg      ends
        End    start
```

۶-۵ ماکروی Inputs

این ماکرو که برای خواندن یک‌رشته از صفحه‌کلید استفاده می‌شود، دارای فرم کلی زیر است:

```
INPUTS      destination, length
```

destination (مقصد) مکانی در دیتا سگمنت است که رشته ورودی در آنجا ذخیره می‌شود و length طول این رشته ورودی را تعیین می‌کند. طول رشته باید بین ۲ تا ۸۰ کاراکتر باشد. کاراکترهای واردشده در متغیر destination به ترتیب ذخیره می‌شوند. با زدن کلید Enter کار ورود اطلاعات خاتمه یافته و در آخر رشته ذخیره‌شده، کاراکتر NULL قرار می‌گیرد. و تعداد کاراکترهای ورودی در ثبات CX ریخته می‌شود.

توجه کنید که در این ماکرو، کد کلید Enter (یعنی کد 13) در متغیر رشته‌ای ذخیره نمی‌گردد. اگر بخواهید بیشتر از تعداد length کاراکتر وارد کنید کامپیوتر بوق زده و آن‌ها را نمی‌گیرد. هنگام ورود، رشته موردنظر روی مانیتور نشان داده می‌شود.

مثال) پس از تعریف و اجرای دستور زیر:

```
.
.
.
STR db      20 dup (?)
.
.
.
Inputs      STR, 20
```

اگر در زمان اجرا عبارت AHMAD↵ را وارد کنید، شکل متغیر STR به‌صورت زیر درمی‌آید:

A	H	M	A	D	\0		...	
STR	STR+1							STR+19

در این حال ثبات CX نیز به‌صورت خودکار برابر 5 می‌شود. در STR حداکثر رشته‌ای به طول 19 کاراکتر را می‌توانیم ذخیره کنیم، چراکه یک‌خانه را برای NULL آخر رشته نیاز داریم (مانند زبان C).

تذکر:

- این ماکرو فقط روی ثبات CX اثر می‌گذارد و ثبات Flag را تغییر نمی‌دهد.

مثال) برنامه‌ای بنویسید که با استفاده از ماکروها نام شمارا گرفته، همان نام را دوباره چاپ کند و قبل از آن پیام Hello را قرار دهد:

```
Include io.h
Sseg segment stack
      DW     64 DUP (?)
Sseg ends
Dseg segment
      Msg    DB "Hello ", 0
      STR    DB 20 Dup (?)
Dseg ends
Cseg segment
      Assume cs: cseg, ds: dseg
Start:mov              ax, seg dseg
            Mov            ds, ax
            Inputs         STR, 20
            Output  msg
            Output  STR
            Mov            ax, 4c00h
            Int            21h
Cseg ends
      End   start
```

۶-۶ ماکروی Inputc

این ماکرو که پارامتر ورودی ندارد و به‌صورت Inputc خالی استفاده می‌شود، یک کاراکتر را از صفحه‌کلید خوانده و کد اسکی آن را در ثبات AL ذخیره می‌کند. در این حالت نیازی به زدن کلید Enter نیست (مشابه تابع Get ch() زبان C). اگر در جواب این ماکرو کلید Enter را بزنید کد 13 در AL ذخیره می‌شود. کاراکتر زده‌شده روی مانیتور نیز نمایش داده می‌شود. این ماکرو شبیه سرویس 1h وقفه 21h عمل می‌کند.

تذکر:

- این ماکرو ثبات AH را نیز صفر می‌کند. پس این ماکرو فقط روی AH و AL اثر می‌گذارد و به ثبات Flag نیز دست نمی‌زند.

فرم کلی استفاده از این ماکرو به‌صورت زیر است:

```
Inputc
```

۶-۷ ماکروی ItoA

ماکروی ItoA (مخفف Integer TO Ascii) یک عدد صحیح (مثبت یا منفی) را به‌صورت 16 بیتی باینری دریافت کرده (مبدأ) و آن را به یک‌رشته 6 کاراکتری معادل در سیستم ده‌دهی تبدیل می‌کند (مقصد). فرم کلی این ماکرو به‌صورت روبرو است:

```
مبدأ و مقصد  Itoa
```

همان‌طور که قبلاً بیان شد، وقفه‌های مربوط به خروجی یا ماکروی output فقط می‌توانند رشته‌ها را نمایش دهند. لذا برای نمایش اعداد صحیح باینری ابتدا به کمک این ماکرو باید آن‌ها را به رشته تبدیل کرد و سپس نمایش داد.

عملوند مبدأ، عدد صریح، یک ثبات یا متغیر است. البته استفاده کردن این ماکرو برای اعداد مشخص و صریح لزومی ندارد. مقصد این ماکرو متغیری رشته‌ای و 6 بایتی است. اگر عدد باینری منفی باشد، یک علامت منفی (-) قبل از رشته عددی در مقصد نوشته می‌شود ولی برای اعداد مثبت علامت (+) نوشته نمی‌شود. اگر طول رشته عددی از 6 کمتر باشد به‌جای رقم‌هایی که وجود ندارند، کاراکترهای فاصله قرار می‌گیرد.

محدوده‌ی رشته عددی ذخیره‌شده در مقصد اعداد صریح 32768- تا 32767 خواهد بود.

تذکر:

- این ماکرو هیچ ثباتی، ازجمله ثبات Flag را تغییر نمی‌دهد.

۸-۶ ماکروی AtoI

ماکروی AtoI (مخفف Ascii TO Integer) برعکس ماکروی ItoA، یک‌رشته عددی ده‌دهی را به باینری تبدیل می‌کند. این ماکرو فقط یک عملوند دارد که همان رشته ورودی به ماکرو است و فرم کلی آن به‌صورت زیر است:

```
Atoi        رشته
```

ماکروی atoi، رشته ورودی را گرفته، تبدیل به باینری کرده و حاصل را در ثبات AX ذخیره می‌کند. رشته ورودی می‌بایست عددی ده‌دهی در محدوده 32768- تا 32767+ باشد. این ماکرو رشته را کاراکتر به کاراکتر از ابتدا می‌خواند و هنگامی‌که به یک کاراکتر غیر از 0 تا 9 برخورد می‌کند عمل تبدیل را تمام می‌کند. مثلاً رشته "32D5" را تبدیل به عدد باینری 32 کرده و به‌صورت زیر در AX ذخیره می‌کند:

AX

0000 0000	0010 0000

تذکر:

- اگر ماکروی atoi عمل تبدیل را به‌صورت موفقیت‌آمیز انجام دهد، پرچم OF برابر صفر می‌شود ولی اگر مثلاً رشته خارج از محدوده 32768- تا 32767+ باشد OF برابر 1 می‌شود. اگر نتیجه حاصل در AX صفر باشد، ZF مساوی یک می‌شود و اگر این عدد منفی باشد SF برابر 1 می‌گردد. این ماکرو CF را نیز صفر می‌کند. پس این ماکرو ثبات AX و Flag را مطابق فوق تغییر می‌دهد.

مثال) برنامه‌ای بنویسید که دو عدد را از ورودی خوانده و جمع آن‌ها را بر روی مانیتور نمایش دهد.

```
   Include io.h
   Cr   EQU 0dh
   Lf   EQU 0ah
; -------------------------------------------------
   Sseg segment stack
      DW    100H DUP (?)
   Sseg ends
; -------------------------------------------------
```

```
Dseg segment
        N1 DW?
        N2 DW?
        Prompt DB 'Enter Numbers: ' cr, lf, 0
        STR     DB 40 Dup (?)
        L1      DB cr, lf,'the sum is:'
        Sum     DB 6 Dup (?)
        DB cr, lf, 0
Dseg ends
; ---------------------------------------------------
Cseg segment
                Assume cs: cseg, ds: dseg
Start:mov       ax, seg dseg
                Mov     ds, ax
                Output prompt
                Inputs  STR, 40
                Atoi    STR
                Mov     N1, ax          ; N1 خواندن اولین عدد و ذخیره باینری معادل آن در متغیر
                Output prompt, 15
                Inputs  STR, 40
                Atoi    STR
                Mov     N2, ax          ; N2 خواندن دومین عدد و ذخیره باینری معادل آن در متغیر
                Mov     ax, N1
                Add     ax, N2                  ; AX ← N1 + N2 جمع دو عدد
                Itoa    sum, ax
                Output L1                       ; چاپ نتیجه خروجی
                Mov     ax, 4c00h
                Int     21h                     ; خروج از برنامه
Cseg ends
        End     start
```

۶-۹ دستورالعمل‌های محاسباتی

انجام اعمال محاسباتی بر روی‌داده‌های دودویی، شامل جمع، تفریق، ضرب و تقسیم داده‌های عددی است. داده‌ها ممکن است علامت‌دار یا بدون علامت باشند. درست است که ما معمولاً از محاسبات ده‌دهی استفاده می‌کنیم ولی محاسبات در کامپیوتر به‌صورت دودویی انجام می‌شود و لذا آشنایی با محاسبات دودویی، ضروری است.

۶-۹-۱ جمع (ADD)

برای جمع دو عدد از دستور ADD بافرم کلی زیر استفاده می‌شود:

ADD<عملوند اول> ,<عملوند دوم>

این دستور عملوند اول را با عملوند دوم جمع کرده و حاصل را در عملوند اول می‌ریزد. یعنی:

<عملوند اول> = <عملوند اول> + <عملوند دوم>

طبق قواعد کلی که برای دستور mov نیز بیان شد عملوند اول می‌تواند ثبات یا متغیر باشد ولی عدد نمی‌تواند باشد.

عملوند دوم می‌تواند ثبات، متغیر یا عدد ثابت باشد. همچنین طبق قاعده کلی اندازه عملوندها می‌بایست هر دو 8 بیتی یا 16 بیتی باشند. همچنین دو عملوند نمی‌توانند متغیر باشند.

<عملوند اول> ← Reg, Memory

<عملوند دوم> ← Reg, Memory, Number

مثال)

دستورات زیر درست می‌باشند	دستورات زیر غلط هستند

ADD AL, BH
SUM DB, 36
ADD CH, SUM
ADD SUM, 5
ADD SUM, DL
ADD CX, DX
	ADD 5, AH
	ADD Item1, Item2
	ADD CX, AH
	SUM DB, 7
	ADD AX, SUM

اثر دستور ADD بر روی پرچم‌ها:

دستور ADD روی پرچم‌های A, P, Z, S, O, C اثر می‌گذارد به این صورت که:

الف) اگر جواب جمع صفر شود پرچم ZF برابر یک می‌گردد در غیر این صورت ZF=0 می‌شود.

ب) اگر جواب منفی باشد (یعنی آخرین بیت سمت چپ جواب "1" باشد) پرچم SF نیز یک می‌شود و در غیر این صورت SF=0 می‌گردد. به‌عبارت‌دیگر بیت سمت چپ جواب در SF ریخته می‌شود.

ج) هنگام جمع، آخرین بیتی که از بایت (در جمع دو عدد 8 بیتی) یا (در جمع دو عدد ۱۶ بیتی خارج می‌شود در پرچم CF ریخته می‌شود.

۶-۹-۲ تفریق

برای تفریق از دستور SUB بافرم کلی زیر استفاده می‌کنیم:

SUB <عملوند اول>, <عملوند دوم>

این دستور عملوند اول را منهای عملوند دوم کرده و حاصل را در عملوند اول می‌ریزد یعنی:

<عملوند اول> = <عملوند اول> – <عملوند دوم>

مشابه دستور ADD در دستور SUB نیز عملوندها می‌بایست هر دو 8 بیتی یا هر دو 16 بیتی باشند. عملوند اول می‌تواند ثبات یا متغیر و عملوند دوم می‌تواند ثبات، متغیر یا عدد باشد. همچنین هر دو عملوند همزمان نمی‌توانند متغیر باشند.

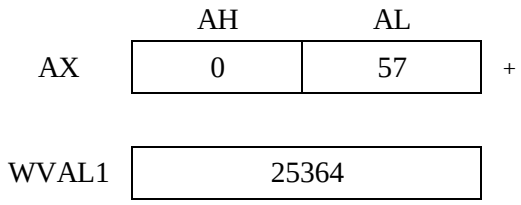
۶-۹-۳ جمع و تفریق یک بایت و یک کلمه (دستور CBW[1])

همان‌طور که قبلاً بیان شد برای جمع و تفریق دو عملوند باید هم‌اندازه باشند. حال اگر بخواهیم عددی 8 بیتی را با عددی 16 بیتی جمع کنیم، ابتدا می‌بایست عدد 8 بیتی را به عددی 16 بیتی تبدیل کرده و سپس اعداد 16 بیتی موجود را باهم جمع کنیم.

اگر اعداد بدون علامت باشند کافی است پشت عدد 8 بیتی، عدد صفر قرار دهید تا 16 بیتی معادل آن به دست آید.

مثال)

```
BVAL    DB    57
WVAL    DW    25364
…
MOV     AL, BVAL
MOV     AH, 0
ADD     AX, WVAL
…
```



ولی اگر اعداد علامت‌دار باشند بایستی برای تبدیل بایت به کلمه از دستور CBW(Convert Byte to Word) استفاده کنیم. این دستور که عملوند ندارد فقط با AL و AH کار می‌کند. این دستور بیت سمت چپی یعنی بیت علامت AL را تست می‌کند، اگر این بیت یک باشد (یعنی AL منفی باشد) AH را با یک پر می‌کند و اگر بیت علامت AL صفر باشد (یعنی AL مثبت باشد) AH را با صفر پر می‌کند. به‌عبارت‌دیگر CBW ثبات AH را بابیت علامت AL پر می‌کند.

مثال) برای جمع متغیر 8 بیتی علامت‌دار BVAL با متغیر 16 بیتی علامت‌دار WVAL به‌صورت زیر عمل می‌کنیم:

```
BVAL    DB    -57
WVAL    DW    25364
…
MOV     AL, BVAL
CBW
ADD     AX, WVAL
```

تذکر:

- دستور CBW روی فلگ ها اثری ندارد.

[1] Convert Byte to Word

۶-۹-۴ دستور CWD[1]

این دستور محتوای ۱۶ بیتی به ۳۲ بیتی تبدیل می‌کند و همچنین به هیچ‌گونه عملوندی نیاز ندارد.
دستور CWD نیز علامت ثبات AX را در ثبات DX تکرار می‌کند و این دستور نیز روی فلگ ها اثری ندارد.

۶-۹-۵ جمع و تفریق اعداد بزرگ (دستورات ADC و SBB)

نحوه جمع و تفریق اعداد بزرگ (مثلاً 32 بیتی) را با مثال زیر نشان می‌دهیم:

مثال) می‌خواهیم اعداد ۳۲ بیتی 3ADE68B1H و 075BCD15H را باهم جمع کنیم. اعداد مذکور را می‌توانیم در ۲ طبقه ۱۶ بیتی مشابه زیر در نظر گرفته و از سمت چپ طبقه به طبقه آن‌ها را با یکدیگر جمع کنیم:

```
  1
  ↖
3ADE    68B1    +
075B    CD15
-----------------
423A    35C6
```

توجه کنید که از طبقه سمت راست یک Carry به طبقه بعدی رفته است.

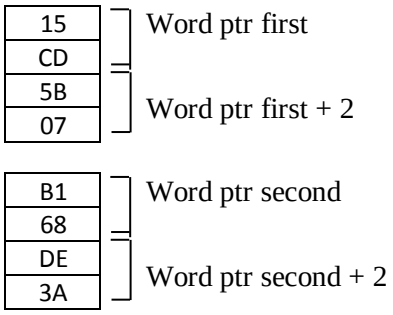
تکه برنامه‌ای که اعمال فوق را انجام دهد به‌صورت زیر است:

(Word ptr: برای اشاره کردن به یک کلمه از یک متغیر دوکلمه‌ای استفاده می‌شود.)

```
(First ← First + secound)
        .
        .
        .
First       DD    075BCD15H
Secound     DD    3ADE68B1H
        .
        .
        .
MOV         AX, word ptr secound        ; عدد 68B1 را در AX می‌ریزد

ADD         word ptr first, AX          ; طبقه اول را باهم جمع می‌کند

MOV         AX, word ptr secound + 2    ; عدد 3ADE را در AX می‌ریزد

ADC         word ptr first + 2, AX      ; طبقه دوم را باهم جمع می‌کند
```

[1] Convert Word Double

با توجه به شکل‌های زیر:



ابتدا کلمه سمت راستی اعداد را با دستور ADD جمع کرده و اگر Carry داشت، این Carry در CF ذخیره می‌شود. سپس کلمه سمت چپی اعداد را با دستور ADC[1] باهم جمع می‌کنیم.

- دستور ADC دقیقاً مانند دستور ADD است مضاف بر این‌که رقم نقلی Carry را نیز جمع می‌کند.
- درصورتی‌که CF=0 باشد: ADD=ADC

به همین ترتیب دستور SBB[2] مشابه SUB بوده ولی CF را هنگام عمل تفریق شرکت می‌دهد. Borrow به معنای قرض است. در عمل تفریق اگر 1 قرضی از طبقه بعد وجود داشته باشد CF برابر 1 می‌شود. در تفریق اعداد بزرگ نیز، اعداد را به‌صورت طبقات 16 بیتی در نظر گرفته و سپس طبقه اول (سمت راستی) را با دستور SUB و طبقات بعدی را با دستور SBB تفریق می‌کنیم.

مثال) عمل تفریق متغیرهای مثال قبلی به‌صورت First ← first – second با دستورات زیر انجام می‌گیرد:

```
Mov ax, word ptr second
Sub  word ptr first, ax
Mov ax, word ptr second + 2
Sbb word ptr first +2, ax
```

تذکر:

- دستورات ADC و SBB روی فلگ های Z, S, P, A, O, C اثر می‌گذارند.

۶-۹-۶ دستور CLC[3]

این دستور CF را پاک می‌کند.

۶-۹-۷ دستورات INC[4] و DEC[5]

دستور INC یک واحد به یک متغیر یا ثبات اضافه کرده و DEC یک واحد کم می‌کند. INC اسمبلی مشابه ++ زبانC و DEC اسمبلی مشابه -- زبانC است.

---

[1] Add with Carry
[2] SuB with Borrow
[3] Clear Carry flag
[4] INcrement
[5] DECrement

فرم کلی این دستورات به‌صورت زیر است:

INC عملوند

DEC عملوند

عملوند ← Register / Memory

یعنی این دستورات فقط یک عملوند داشته و این عملوند می‌تواند ثبات یا متغیر 8 یا 16 بیتی باشند. ازنظر جواب نهایی دستور مثل INC AX مشابه دستور ADD AX, 1 است ولی ازنظر سرعت و مصرف حافظه دستورات INC و DEC بهتر و مؤثرتر از دستورات ADD و SUB هستند. دستورات INC و DEC در حافظه‌ها برای تغییر اندیش حلقه زیاد استفاده می‌شوند.

۶-۹-۸ دستور NEG[1]

دستور NEG از محتوای عملوند خود مکمل ۲ می‌گیرد. یعنی آن را منفی می‌کند. فرم کلی این دستور به‌صورت زیر است:

NEG عملوند

تنها عملوند این دستور می‌تواند ثبات یا متغیر 8 یا 16 بیتی باشد. مثلاً دستوراتی شبیه زیر درست هستند:

```
NEG     AX
NEG     VALUE
```

دستوری مثل NEG 14 غلط است چراکه جلوی NEG عدد صریح نمی‌آید.

تذکر:

- این دستور روی فلگ های P, A, C, O, S, Z اثر می‌گذارد.

مثال) با اجرای دو دستور زیر محتوای ثبات AH برابر 10010010 می‌شود:

```
MOV     AH, 01101110B
NEG     AH
```

۶-۹-۹ دستور NOT

این دستور نقیض (مکمل۱) عملوند موردنظر را در خودش قرار می‌دهد. همانند دستور NEG تک عملوندی است.

عملوند ← Register / Memory

[1] NEGate

۶-۹-۱۰ دستورات ضرب MUL[1] و IMUL[2]

دستور MUL برای ضرب اعداد بدون علامت و دستور IMUL برای ضرب اعداد علامت‌دار استفاده می‌شوند. فرم کلی این دستورات به‌صورت زیر است:

```
Mul          عملوند

عملوند Imul
```

- این دستورات فقط یک عملوند دارند.
- عملوند می‌تواند یک ثبات یا متغیر از نوع 8 بیتی (بایت) یا 16 بیتی (کلمه) باشد.
- اگر عملوند جلوی دستورات فوق 8 بیت باشد، عملوند دوم همواره AL فرض شده و حاصل‌ضرب در AX ریخته می‌شود.
- اگر عملوند جلوی دستورات فوق 16 بیتی باشد، عملوند دوم همواره AX فرض شده و حاصل‌ضرب در جفت ثبات DX:AX ریخته می‌شود.

مثال) تکه برنامه زیر متغیر بدون علامت BVAL که 8 بیتی است را در عدد 30 ضرب می‌کند:

```
BVAL        DB      20
.
.
.
MOV         AL, 30
MUL                 BVAL            ;       AX ← BVAL × AL
```

در تکه برنامه فوق چون BVAL بایت است در AL ضرب شده و حاصل این ضرب یعنی عدد 600 در AX ریخته می‌شود. وقتی بایتی (8 بیت) در بایتی (8 بیت) ضرب می‌شود، حاصل‌ضرب حداکثر یک کلمه (16 بیتی) است و وقتی کلمه‌ای در کلمه‌ای ضرب می‌شود حاصل‌ضرب حداکثر ۲ کلمه (32 بیتی) است.

مثال) تکه برنامه زیر متغیر علامت‌دار BVAL که 8 بیتی است را در عدد 30 ضرب می‌کند:

```
BVAL        DB      -20
.
.
.
MOV         AL, 30
IMUL BVAL
```

مثال) تکه برنامه زیر متغیر 16 بیتی بدون علامت WVAL را در عدد D903H ضرب می‌کند:

```
WVAL        DW      3E8H
MOV         AX, 0D903H
MUL                 WVAL                    ;       DX: AX ← WVAL × AX
```

[1] Multiply
[2] Integer Multiply

منظور از DX:AX این است که AX در سمت راست DX قرارگرفته و باهم ظرفی 32 بیتی را تشکیل می‌دهند:

	DX	AX			DX	AX
قبل از ضرب	????	D903	⇨	بعد از ضرب	034F	B3B8

تذکر:

- MUL و IMUL فقط روی CF و OF اثر بامعنی می‌گذارند و در برنامه‌نویسی فقط این بیت‌ها را در نظر می‌گیریم. اثر این دستورات بر روی Z, S, P, A تعریف‌نشده و نامشخص است.
- اکثر اوقات حاصل‌ضرب آن‌قدر طولانی نیست که نیاز به AH یا DX باشد. برای تشخیص این مطلب می‌توان از پرچم‌های CF و OF به‌صورت زیر استفاده کرد:
  - اگر CF=0 و OF=0 باشد، AH در ضرب بایت‌ها و DX در ضرب کلمه‌ها لازم نیستند.
  - اگر CF=1 و OF=1 باشد، AH در ضرب بایت‌ها و DX در ضرب کلمه‌ها لازم هستند.

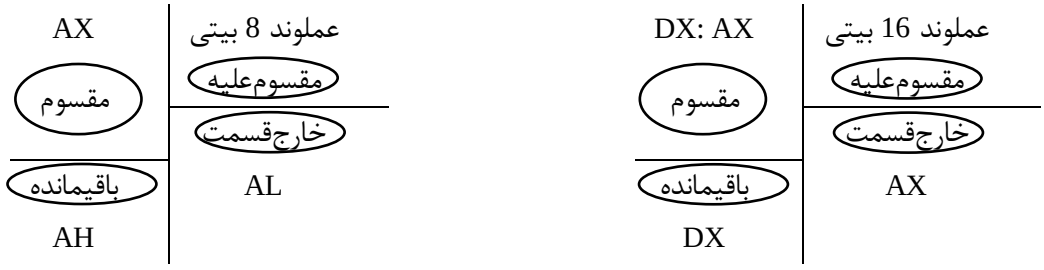
۶-۹-۱۱ دستورات تقسیم DIV[1] و IDIV[2]

دستور DIV برای تقسیم اعداد بدون علامت و دستور IDIV برای تقسیم اعداد علامت‌دار استفاده می‌شوند. فرم کلی هر دو دستور فقط یک عملوند دارد که این عملوند مقسوم‌علیه تقسیم است:

DIV	**عملوند**

IDIV	**عملوند**

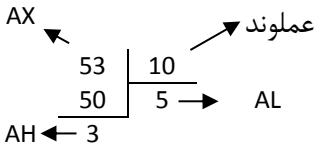
- می‌توان یک کلمه (16بیت) را بر یک بایت (8بیت) یا دو کلمه (32بیت) را بر یک کلمه (16بیت) تقسیم کرد.
- اگر عملوند (مقسوم‌علیه) یک بایتی (8 بیتی) باشد، DIV و IDIV فرض می‌کنند ثبات AX مقسوم است و خارج‌قسمت در AL و باقیمانده در AH قرار می‌گیرد.
- اگر عملوند (مقسوم‌علیه) یک‌کلمه‌ای (16 بیتی) باشد، DIV و IDIV فرض می‌کنند DX:AX حاوی مقسوم‌علیه و خارج‌قسمت در AX و باقیمانده در DX قرار خواهد گرفت:



[1] Divide
[2] Integer Division

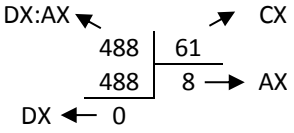
مثال) که برنامه زیر عدد 53 را بر 10 تقسیم کرده و پس از انجام این عمل، مقدار AL=5 و AH=3 خواهد شد:

```
BVAL        DB      10
.
.
.
Mov ax, 53
Div  BVAL
```



مثال)

```
MOV    DX, 0000H
MOV    AX, 488
MOV    CX, 61
IDIV     CX
```



تذکر:

- مشابه زبان‌های C و پاسکال در زبان اسمبلی نیز علامت باقیمانده همواره مساوی علامت مقسوم‌علیه است و علامت خارج‌قسمت از ضرب علائم مقسوم و مقسوم‌علیه به دست می‌آید.

مثلاً در پاسکال داریم:

```
13 div 4 = 3          ,        13 mod 4 = 1
13 div -4 = -3        ,        13 mod -4 = 1
-13 div 4 = -3        ,        -13 mod 4 = -1
-13 div -4 = -3       ,        -13 mod -4 = -1
```

- دستورات تقسیم هیچ‌گونه مقادیر مفیدی در فلگ ها قرار نمی‌دهند و ممکن است مقادیر قبلی پرچم‌های ZF, SF, PF, AF, OF, CF را پاک کنند به‌عبارت‌دیگر اثر دستورات تقسیم‌بر روی فلگ ها تعریف‌نشده است.

مثال) با استفاده از ماکروها برنامه‌ای بنویسید که مقدار x را دریافت کرده و مقدار Y را برحسب معادله زیر محاسبه و در صفحه‌نمایش نشان دهد.

$Y=3X^2-2X+5$

```
Include io.h
Datasg segment
      Msg1 db "Enter x:", 0
      Msg2 db "y=", 0
      X db 10 dup (?)
      Y db 10 dup (?)
Datasg ends
```

```
Codsg segment
        Assume cs: codsg, ds: datasg
Start:  mov ax, datasg
        Mov ds, ax

        Output msg1             ; msg1 چاپ

        Inputs x, 8             ; x دریافت رشته

        Atoi x                  ; تبدیل رشته x به عدد

        Mov bx, ax
        Mul ax                  ; x2 محاسبه

        Mov cx, 3
        Mul cx                  ; 3x  محاسبه

        Mov dx, ax

        Sub dx, bx
        Sub dx, bx              ; -2x محاسبه

        Add dx, 5               ; +5 محاسبه

        Output msg2             ; msg2 چاپ
        Itoa y, dx              ; y تبدیل عدد dx به رشته
        Output y                ; Y چاپ

        Mov ax, 4c00h
        Int 21h
Codsg ends
    End start
```

فصل هفتم

دستورات کنترلی(پرش) و پیاده‌سازی ساختارها

در برنامه‌هایی که تاکنون نوشته شدند، دستورات برنامه به ترتیب از بالا به پایین اجرا می‌شوند. یعنی شیوه اجرای دستورات، ترتیبی است، اما در همه برنامه‌ها این کار امکان‌پذیر نیست. گاهی ممکن است بر اساس شرایطی بخواهیم تعدادی از دستورات اجرا شوند و از اجرای تعداد دیگری از دستورات صرف‌نظر گردد. در این موارد باید از دستوراتی استفاده کنیم که امکان اجرای بعضی از دستورات و عدم اجرای دستورات دیگر را فراهم کنند. این دستورات را **ساختارهای تصمیم** می‌گویند.

گاهی لازم است، بر اساس شرایطی، یک یا چند دستور، به‌جای اینکه یک‌بار اجرا شوند، چندین بار اجرا شوند و درنتیجه حلقه‌های تکرار را به وجود آورند. امکاناتی که این شیوه اجرای دستورات را فراهم می‌کنند، **ساختارهای تکرار** نام دارند.

۷-۱ انواع آدرس‌ها

یکی از مواردی که در انتقال برنامه از نقطه‌ای به نقطه دیگر باید موردتوجه قرار گیرد، شیوه‌های آدرس‌دهی است. توجه به شیوه‌های آدرس‌دهی موجب می‌شود تا برنامه‌نویس با خطای آدرس‌دهی مواجه نشود. سه نوع آدرس در زبان اسمبلی وجود دارد که عبارت‌اند از:

۷-۱-۱ آدرس کوتاه[1]

در آدرس کوتاه، می‌توان فاصله ۱۲۸- تا ۱۲۷ بایت را آدرس‌دهی کرد.

۷-۱-۲ آدرس نزدیک[2]

در آدرس نزدیک، فاصله ۳۲۷۶۷- تا ۳۲۷۶۷ بایت را در داخل یک سگمنت می‌توان آدرس‌دهی کرد.

۷-۱-۳ آدرس دور[3]

در آدرس دور می‌توان تا فاصله ۳۲ کیلوبایت را در سگمنت دیگر نیز آدرس‌دهی کرد.

تذکر:

- در آدرس کوتاه، از یک آفستِ یک بایتی و در آدرس نزدیک، از آفست یک یا دوکلمه‌ای می‌توان استفاده کرد. آدرس دور نیز با آدرس سگمنت و آدرس آفست در آن سگمنت، قابل‌دستیابی است.
- انواع آدرس‌دهی‌ها بر ثبات IP و آدرس‌دهی دور بر ثبات CS نیز اثر می‌گذارد.
- توجه داشته باشید که آدرس‌دهی‌هایی که به‌طرف بالای برنامه انجام می‌شود، آدرس منفی و آدرس‌دهی‌هایی که به‌طرف پایین برنامه انجام می‌شود، آدرس مثبت محسوب می‌شوند.

---

[1] Short
[2] Near
[3] Far

۷-۲ مفهوم پرش

برای اینکه ساختارهای تصمیمی را پیاده‌سازی کنیم، باید بتوانیم از نقطه از برنامه به نقطه دیگری پرش کنیم و اجرای برنامه را از نقطه جدید ادامه بدهیم. برای آشنایی با مفهوم پرش، الگوریتم زیر را در نظر بگیرید:

الگوریتم نمونه:

الگوریتمی که ۵ عدد را از ورودی خوانده، میانگین آن ۵ عدد را محاسبه کرده، به خروجی می‌برد. متغیرهایی که در الگوریتم استفاده‌شده‌اند عبارت‌اند از:

I: شمارنده اعداد، SUM: مجموع اعداد، AVE: میانگین اعداد و X: عددی که خوانده می‌شود.

```
۱- I ← 1
۲- SUM ← 0
۳- اگر I > 5 برو به ۸
۴- X را بخوان
۵- SUM ← SUM + X
۶- I ← I + 1
۷- برو به دستور ۳
۸- AVE ← SUM / 5
۹- AVE را چاپ کن
```

در این الگوریتم، در دستور ۳ مقدار I با 5 مقایسه شد و درصورتی‌که I > 5 باشد، کنترل الگوریتم به دستور ۸ می‌رود. در اینجا یک عمل پرش از دستور ۳ به دستور ۸ انجام می‌شود. همان‌طور که ملاحظه می‌شود، قبل از انجام پرش، باید شرطی تست شود. این‌گونه پرش‌ها را **پرش شرطی** گویند. اما دستور ۷ دستور پرش دیگری است که در آن بدون هیچ‌گونه شرطی، کنترل الگوریتم به دستور ۳ می‌رود، این پرش را **پرش غیرشرطی** گویند. دستورات بین دستور شماره ۳ و دستور شماره ۸، به تعداد ۵ بار تکرار می‌شوند و لذا تشکیل **حلقه تکرار** را می‌دهند. بعضی از شماره‌هایی که کنار دستورات وجود دارد، ضروری‌اند، به‌عنوان‌مثال، برای اینکه بتوانیم از دستور ۳ به دستور ۸ پرش کنیم، این دستور حتماً باید شماره داشته باشد. پرش از دستور شماره ۸ به دستور شماره ۳ ایجاب می‌کند که دستور شماره ۳ حتماً با شماره‌ای مشخص شود. این شماره‌ها را **برچسب** می‌گوییم.

در زبان اسمبلی، به هر دستوری که بعداً مراجعه می‌شود باید برچسبی را نسبت داد. برچسب در زبان اسمبلی یک شناسه است که به کولن (:) ختم می‌شود، مثل P1: ، Label1: و P2: .

۷-۳ پرش‌های غیرشرطی

۷-۳-۱ دستور JMP[1]

دستور JMP برای پرش به یک خط موردنظر استفاده می‌شود و برای پیاده‌سازی پرش غیرشرطی، در زبان اسمبلی از دستور JMP استفاده می‌شود. این دستور مشابه goto در زبان C و پاسکال است. جلوی این دستور نام یک برچسب (Label) می‌آید و در خطی که قرار است پرش به آن صورت گیرد همان برچسب همراه با علامت (:)

[1] Jump

نوشته می‌شود. این دستور، کنترل اجرای برنامه را تحت هر شرایطی از نقطه‌ای به نقطه دیگر منتقل می‌کند. دستور JMP به‌صورت زیر به کار می‌رود:

```
JMP <برچسب دستور>
```

اجرای دستور JMP باعث می‌شود که کنترل اجرای برنامه به دستوری برود که با <برچسب دستور> ، مشخص می‌شود. به‌عنوان‌مثال، در دستورات زیر، JMP باعث انتقال کنترل به دستوری با برچسب P1 می‌شود:

```
        …
        JMP   P1
        …
        …
        …
P1:     MOV   AX, 1
        …
```

در مورد دستور JMP باید به آدرس‌های کوتاه، نزدیک و دور توجه کنید. وقتی اسمبلر درگذر اول برنامه را ترجمه می‌کند، طول هر دستورالعمل را محاسبه می‌نماید تا بتواند بر اساس طول دستورالعمل‌ها، حافظه لازم را اختصاص دهد. طول دستورالعمل JMP بستگی به این دارد که پرش آن کوتاه، نزدیک یا دور باشد. اگر برچسبی که JMP به آن پرش می‌کند در محدوده آدرس ۱۲۸- تا ۱۲۷ باشد، این پرش، کوتاه است و طول دستور JMP دو بایتی است (یک بایت برای عملوند و یک بایت برای نوع عمل). عملوند این دستور، به‌عنوان آفستی است که در موقع اجرای برنامه به ثبات IP اضافه می‌شود تا به دستور موردنظر مراجعه گردد.

اگر برچسبی که JMP به آن مراجعه می‌کند، از ۱۲۸- تا ۱۲۷ تجاوز نکند، در محدوده 32K به‌عنوان پرش نزدیک منظور می‌شود و در این صورت JMP، ۳ بایتی است (یک بایت برای نوع عمل و دو بایت برای عملوند).

با استفاده از عملگر SHORT، NEAR PTR و FAR PTR نیز می‌توان انواع مختلف دستور JMP را مشخص کرد. به‌عنوان‌مثال، دستورات زیر، کاربرد SHORT را نشان می‌دهند:

```
Labrl1:   JMP   short   label2  ;       پرش کوتاه
          .
          .
          .
Label2:   JMP   label1  ;       پرش کوتاه
```

اکنون دستورات زیر را در نظر بگیرید:

```
          JMP   near    ptr     L1      ;       پرش نزدیک
          .
          .
          .
L1:
          .
          .
          .
          JMP   far     ptr     L2      ;       پرش دور
          .
          .
          .
L2:
```

تذکر:

اسمبلرهای مورداستفاده معمولاً دو گذره هستند که درگذر اول، تمام برنامه را خوانده و در جدول نماد، اسامی و برچسب‌های استفاده‌شده را نگهداری می‌کند و درگذر دوم از جدول نماد گذر اول استفاده می‌شود.

مثال) برنامه‌ای بنویسید که مقدار صفر را در ثبات‌های AX، BX و CX قرار می‌دهد و سپس یک واحد به هر یک از این ثبات‌ها اضافه می‌کند و این عمل را به‌طور نامحدود ادامه می‌دهد.

در این برنامه، چون با دستور JMP (پرش غیرشرطی) به برچسب P1 مراجعه می‌شود، برنامه هیچ‌گاه خاتمه نمی‌یابد. به‌عبارت‌دیگر، حلقه تکرار بی‌نهایت ایجاد می‌شود.

```
Stacksg     segment     stack
    Db      32    dup   (?)
Stacksg     ends

Codseg      segment     para    'code'
    Main    proc  far
            Assume ss: stacksg,     cs: Codseg

            Mov   ax, 0
            Mov   bx, 0
            Mov   cx, 0
    P1:     Inc   ax                    شروع حلقه
            Inc   bx
            Inc   cx
            Jmp   P1                    پایان حلقه
            Ret
    Main    endp

Codseg      ends
    End     main
```

۴-۷ پرش‌های شرطی

در پرش‌های شرطی، لازم است مقایسه‌ای صورت پذیرد و بر اساس نتایج مقایسه، عمل پرش انجام شود. دستورالعمل‌های زیادی برای انجام پرش وجود دارند. ابتدا روش انجام مقایسه را بررسی کرده، سپس به دستورات پرش‌های شرطی می‌پردازیم.

۱-۴-۷ دستور CMP[1]

برای انجام مقایسه و بررسی شرط‌ها از دستور CMP استفاده می‌شود. این دستور به‌صورت زیر به کار می‌رود:

CMP <عملوند۱> و <عملوند۲>

عملوندهای دستوری CMP می‌توانند به‌صورت های زیر باشند:

[1] CoMpare

عملوند۱	ثبات	حافظه	ثبات	حافظه
عملوند۲	حافظه	ثبات	مقدار ثابت	مقدار ثابت

دستور CMP <عملوند۲> را از <عملوند۱> تفریق می‌کند (مثل دستور SUB)، ولی حاصل تفریق را در <عملوند۱> قرار نمی‌دهد (برخلاف SUB، عملوند را تغییر نمی‌دهد). پس از کجا می‌توان فهمید که نتیجه مقایسه چه شده است؟ دستور CMP پس از انجام مقایسه، فلگ های[1] مختلف ثبات وضعیت را مقداردهی می‌کند و دستورات پرش شرطی که در ادامه بحث می‌شوند، با توجه به مقدار فلگ ها، عمل پرش را انجام می‌دهند. این دستور به مقادیر عملوند۱ و عملوند۲ دست نمی‌زند ولی روی پرچم‌های AF، PF، SF، ZF، CF و OF اثر می‌گذارد. به‌عنوان‌مثال، دستور زیر را در نظر بگیرید:

```
CMP        DX, 0
```

اگر ثبات DX برابر با 0 باشد، دستور CMP فلگ ZF را برابر با 1 قرار می‌دهد. بلافاصله پس از دستور CMP باید دستورات پرش شرطی را قرارداد تا بر اساس مقدار فلگ ZF یا هر فلگ دیگری که تغییر می‌کنند، کنترل اجرای برنامه به نقطه دلخواهی منتقل شود. دستورات پرشی که پس از دستور CMP مورداستفاده قرار می‌گیرند.

نکته:

دستورات پرش که پس از دستور CMP استفاده می‌شوند به علامت‌دار یا بدون علامت بودن داده‌های مقایسه شده بستگی دارند. در داده‌های عددی علامت‌دار بیت سمت چپ، بیت علامت است که منفی یا مثبت بودن را نشان می‌دهد و نباید در مقایسه شرکت کنند.

به‌عنوان‌مثال فرض کنید ثبات CX حاوی 10000110b و ثبات DX حاوی 00010110b است. اکنون دستور زیر را در نظر بگیرید:

```
CMP        CX, DX
```

با این دستور، محتویات CX با محتویات DX مقایسه می‌شود و نتیجه آن فلگ ها را مقدار می‌دهد. در اینجا اگر داده‌ها را علامت‌دار در نظر بگیرید، چون CX منفی است (بیت سمت چپ ۱ است)، از DX کوچک‌تر است ولی اگر داده‌ها را بدون علامت در نظر بگیرید، مقدار CX بزرگ‌تر از مقدار DX است.

مثال)

CX: 10000110	CX<DX	بدون علامت
DX: 00011010	CX>DX	علامت‌دار

در اینجا نیز باید دستور مناسبی را برای انجام پرش شرطی انتخاب کرد:

[1] Flags

۷-۴-۱-۱ پرش‌های شرطی برای داده‌های بدون علامت

برای داده‌های بدون علامت، از دستورات خاصی استفاده می‌شود که در جدول زیر آمده‌اند:

دستور	عملکرد	فلگ هایی که تست می‌شوند
JE[1] یا JZ[2]	پرش در صورت مساوی بودن، یا پرش در صورت صفر بودن	ZF
JNE[3] یا JNZ[4]	پرش در صورت عدم‌تساوی، یا پرش در صورت صفر نبودن	ZF
JA[5] یا JNBE[6]	پرش در صورت بیشتر بودن، یا پرش در صورت کمتر یا مساوی نبودن	CF و ZF
JAE یا JNB	پرش در صورت بیشتر یا مساوی بودن، یا پرش در صورت کمتر نبودن	CF
JB یا JNAE	پرش در صورت کمتر بودن، یا پرش در صورت بیشتر یا مساوی نبودن	CF
JBE یا JNA	پرش در صورت کمتر یا مساوی بودن، یا پرش در صورت بیشتر نبودن	CF

۷-۴-۱-۲ دستورات پرش شرطی برای داده‌های علامت‌دار

دستورات پرش شرطی خاصی برای داده‌های علامت‌دار وجود دارند، این دستورات در جدول زیر آمده‌اند:

دستور	عملکرد	فلگ هایی که تست می‌شوند
JE یا JZ	پرش در صورت مساوی بودن، یا پرش در صورت صفر بودن	ZF
JNE یا JNZ	پرش در صورت مساوی نبودن، یا پرش در صورت صفر نبودن	ZF
JG[7] یا JNLE[8]	پرش در صورت بزرگ‌تر بودن، یا پرش در صورت کوچک‌تر یا مساوی نبودن	SF ،OF و ZF
JGE[9] یا JNL[10]	پرش در صورت بزرگ‌تر یا مساوی بودن، یا پرش در صورت کوچک‌تر نبودن	SF ،OF
JL یا JNGE	پرش در صورت کوچک‌تر بودن، یا پرش در صورت بزرگ‌تر یا مساوی نبودن	SF ،OF
JLE یا JNG	پرش در صورت کوچک‌تر یا مساوی بودن، یا پرش در صورت بزرگ‌تر نبودن	SF ،OF و ZF

۷-۴-۱-۳ دستورات پرش خاص

تعدادی از این دستورات پرشی وجود دارند که تست‌های خاصی را انجام می‌دهند. این دستورات در جدول زیر آمده‌اند:

---

[1] Jump Equal
[2] Jump Zero
[3] Jump Not Equal
[4] Jump Not Zero
[5] Jump Above
[6] Jump Not Below or Equal
[7] Jump Greater
[8] Jump Not Less or Equal
[9] Jump Greater or Equal
[10] Jump Not Less

دستور	عملکرد	فلگ هایی که تست می‌شوند
JCXZ[1]	پرش درصورتی‌که CX برابر صفر باشد	هیچ‌کدام
JC[2]	پرش درصورتی‌که رقم نقلی وجود داشته باشد (مثل JB)	CF
JNC	پرش درصورتی‌که رقم نقلی وجود نداشته باشد	CF
JO[3]	پرش درصورتی‌که سرریز وجود داشته باشد	OF
JNO	پرش درصورتی‌که سرریز وجود نداشته باشد	OF
JPE یا JP[4]	پرش در حالت توازن، یا پرش در حالت توان زوج	PF
JPO یا JNP	پرش در حالت عدم توازن، یا پرش در حالت توازن فرد	PF
JS[5]	پرش در حالت منفی	SF
JNS	پرش در حالت مثبت	SF

دستور JCXZ که در جدول آمده است، ثبات CX را تست می‌کند و درصورتی‌که این ثبات صفر باشد، عمل پرش را انجام می‌دهد. بنابراین لازم نیست این دستور حتماً بعد از یک عمل مقایسه یا حسابی اجرا شود. یکی از کاربردهای این دستور در حلقه‌های تکرار است که در ادامه مورداستفاده قرار می‌گیرد.

کاربرد دستوراتی که در جدول‌های بالاآمده‌اند با تمرین و تجربه زیاد، به‌صورت عادی درمی‌آید. فعلاً لازم نیست که سعی کنید همه آن‌ها را حفظ کنید. دقت داشته باشید که دستوراتی که برای داده‌های علامت‌دار به کار می‌روند، به‌صورت **بزرگ‌تر یا کوچک‌تر بودن** هستند، درحالی‌که دستورات مربوط به داده‌های بدون علامت، به‌صورت **بیشتر یا کمتر بودن** به کار می‌روند.

چند نمونه ساده از کاربرد دستورات:

- ثبات AX حاوی عددی بدون علامت است. درصورتی‌که محتویاتش بیش از ۵۰ باشد، کنترلِ برنامه به دستوری با برچسب P1 می‌رود:

```
CMP       AX, 50
JA   P1
.
.
.
P1:  SUB    AX, AX
```

- ثبات AX حاوی عددی بدون علامت است. درصورتی‌که محتویاتش برابر ۵۰ باشد، کنترل برنامه به دستوری با برچسب P1 می‌رود:

```
CMP       AX, 50
JE   P1
.
.
.
P1:  SUB    AX, AX
```

[1] Jump CX Zero
[2] Jump Carry
[3] Jump OverFlow
[4] Jump Parity
[5] Jump Sign

- ثبات AX حاوی عدد علامت‌دار است درصورتی‌که محتویاتش بزرگ‌تر یا مساوی ۵۰ باشد، کنترل برنامه به دستوری با برچسب P1 می‌رود:

```
CMP       AX, 50
JGE  P1
.
.
.
.
P1:  SUB    AX, AX
```

ثبات AX حاوی عددی علامت‌دار است. درصورتی‌که محتویات آن بزرگ‌تر از ۱۰۰ باشد، کنترل اجرای برنامه به دستوری با برچسب P1 می‌رود:

```
CMP       AX, 100
JG   P1
.
.
.
.
P1:  SUB    AX, AX
```

دقت داشته باشید که در پردازنده‌های 80286 و قبل از آن پرش‌های غیرشرطی با پرشی کوتاه امکان‌پذیر هستند (محدوده پرش از ۱۲۸- تا ۱۲۷ بایت است). درحالی‌که در پردازنده‌های 80386 و به بالا، از آفست‌های ۳۲ بیتی استفاده می‌شود و درنتیجه آدرس‌دهی در محدوده 32K امکان‌پذیر است.

مثال) برنامه‌ای که رشته‌ای را از ورودی خوانده، آن را کاراکتر به کاراکتر به خروجی می‌برد. هدف از این برنامه آشنایی با انتقال کنترل شرطی و ایجاد حلقه تکرار به کمک دستورات شرطی است.

توضیحات:

در این برنامه، لیستی به نام String برای دریافت رشته از ورودی تعریف شد که اجزای آن عبارت‌اند از: max حداکثر طول رشته، len طول واقعی رشته، buffer محل ذخیره رشته. ابتدا صفحه‌نمایش پاک‌شده، مکان‌نما به سطر و ستون خاصی منتقل‌شده، پیامی صادرشده، رشته‌ای از ورودی خوانده می‌شود. پس از خواندن رشته، مکان‌نما به محل مناسبی رفته، در یک حلقه تکرار، کاراکترهای رشته، یکی‌یکی به خروجی منتقل می‌شوند. برای ایجاد حلقه تکرار، از ثبات CX به‌عنوان شمارنده حلقه استفاده‌شده است. مقدار اولیه آن صفر است و پس از خروجی هر کاراکتر، یک واحد به آن اضافه می‌شود و این روند تا رسیدن CL به len (طول واقعی رشته) ادامه دارد. پس‌ازاینکه مقدار CL به len رسید، حلقه تکرار خاتمه می‌یابد. توجه داشته باشید که برای دسترسی به کاراکترهای رشته، ابتدا آدرس آفست رشته را در ثبات BX قراردادیم و هربار یک واحد به آدرس آن اضافه کردیم. برای خروجی کاراکتر نیز از سرویس 02h وقفه 21h استفاده‌شده است.

```
Datasg   segment para 'code'

   Msg1     db      'Enter a string: ',"$"
   Msg2     db      'you entered this string: ',"$"
```

```
    Strlist label       byte
    Max         db      20
    Len         db      ?
    Buffer db   20 dup(' ')
    Dolar       db      '$'
Datasg  ends

        Codesg  segment para 'code'

        Main    proc    far

                Assume  ds: datasg, cs: codesg
                Mov     ax, datasg
                Mov     ds, ax

                Mov     al, 25          ;   تعداد سطر
                Mov     ch, 0
                Mov     cl, 0
                Mov     dh, 24          ; سطر
                Mov     dl, 79          ; ستون
                Mov     bh, 7           ;رنگ
                Mov     ah, 6h
                Int     10h

                Mov     dh, 10          ; سطر
                Mov     dl, 30          ; ستون
                Mov     bh, 0           ; شماره صفحه
                Mov     ah, 2h
                Int     10h

                Lea     dx, msg1
                Mov     ah, 9h
                Int     21h

                Mov     ah, 0ah
                Lea     dx, strlist
                Int     21h

                Mov     dh, 12          ; سطر
                Mov     dl, 30          ; ستون
                Mov     bh, 0           ; شماره صفحه
                Mov     ah, 2h
                Int     10h

                Mov     dx, offset msg2
                Mov     ah, 9h
                Int     21h
```

تعریف محلی برای رشته

پاک کردن صفحه نمایش

انتقال مکان نما

چاپ پیام msg1

دریافت رشته

انتقال مکان نما

چاپ پیام msg2

```
;       چاپ رشته به‌صورت کاراکتر به کاراکتر

                Lea     bx, buffer              ;       آدرس رشته در BX قرار می‌گیرد
                Mov     cl, 0
p1:             Mov     dl, [bx]
                Mov     ah, 2h
                Int     21h
                Inc     bx                      ایجاد حلقه برای چاپ رشته به صورت کاراکتر به کاراکتر
                Inc     cl
                Cmp     cl, len
                Jnz     p1

                Mov     ax, 4c00h
                Int     21h
Mainendp
Codesg  ends
                End     main
```

مثال) برنامه‌ای که رشته‌ای را از ورودی خوانده، آن را به‌طور معکوس، به خروجی می‌برد. هدف از این برنامه، آشنایی با انتقال کنترل شرطی و ایجاد حلقه‌های تکرار به کمک آن است:

برای این‌که رشته ورودی به‌صورت کاراکتر به کاراکتر و از آخرین کاراکتر به اولین کاراکتر چاپ شود، با دستور LEA آدرس آفست Buffer (محل حاوی رشته) در ثبات BX قرار گرفت و برای به دست آوردن انتهای این رشته، طول واقعی رشته یعنی len به ثبات BX اضافه شد. ولی دقت داشته باشید که اگر طول رشته len باشد، در موقعیت buffer+len، کاراکتر 0dh قرار دارد که کد کلید Enter است، به همین دلیل پس از قرار دادن len در ثبات CL(CX)، از آن‌یک واحد کم شده و نتیجه حاصل به BX اضافه‌شده است. بدین ترتیب، BX به آخرین کاراکتر ورودی اشاره می‌کند. این کاراکتر به خروجی منتقل می‌شود و سپس یک واحد از BX کم می‌شود تا به کاراکتر قبلی اشاره کند. این روند تا خروجی تمام کاراکترها ادامه می‌یابد. حلقه تکرار نیز با استفاده از طول رشته (len) و ثبات CX ایجاد می‌شود.

```
Datasg  segment para 'code'
   Msg1         db      'Enter a string: ',"$"
   Msg2         db      'Inverse string is: ',"$"

   Strlist label        byte    ; شروع لیست پارامتر
   Max          db      20                      تعریف محلی برای
   Len          db      ?                       رشته
   Buffer db    20 dup(' ')
   Dolar        db      '$'
Datasg  ends

Codesg  segment para 'code'
Mainproc        far
                Assume  ds: datasg, cs: codesg
                Mov     ax, datasg
                Mov     ds, ax
```

```
        Mov     al, 25          ;  تعداد سطر
        Mov     ch, 0
        Mov     cl, 0
        Mov     dh, 24          ; سطر
        Mov     dl, 79          ; ستون
        Mov     bh, 7           ;رنگ
        Mov     ah, 6h
        Int     10h                                  } پاک کردن صفحه نمایش

        Mov     dh, 10  ; سطر
        Mov     dl, 30          ; ستون
        Mov     bh, 0           ; شماره صفحه
        Mov     ah, 2h
        Int     10h                                  } انتقال مکان نما

        Lea     dx, msg1
        Mov     ah, 9h
        Int     21h                                  } چاپ پیام msg1

        Mov     ah, 0ah
        Lea     dx, strlist
        Int     21h                                  } دریافت رشته

        Mov     dh, 12          ; سطر
        Mov     dl, 30          ; ستون
        Mov     bh, 0           ; شماره صفحه
        Mov     ah, 2h
        Int     10h                                  } انتقال مکان نما

        Mov     dx, offset msg2
        Mov     ah, 9h
        Int     21h                                  } چاپ پیام msg2

        Lea     bx, buffer              ;       آدرس رشته در BX قرار می‌گیرد
        Mov     ch, 0
        Mov     cl, len
        Sub     cl, 1
        Add     bx, cx
        Mov     cl, 0
p1:     Mov     dl, [bx]
        Mov     ah, 2h
        Int     21h                             } چاپ رشته به طور معکوس و به صورت کاراکتر به کاراکتر
        Inc     cl
        Dec     bx
        Cmp     cl, len
        Jnz     p1
```

```
            Mov     ax, 4c00h
            Int     21h
Mainendp
Codesg  ends
            End     main
```

مثال از خروجی برنامه:

```
Enter a string: ABCD↵
Inverse string is: DCBA
```

مثال) برنامه‌ای که رشته‌ای را از ورودی خوانده، سپس دو کاراکتر CH1 و CH2 را نیز از ورودی می‌خواند و تمام کاراکترهای CH1 را در رشته خوانده‌شده، به CH2 تبدیل می‌کند.

توضیحات:

ابتدا یک‌رشته و سپس دو کاراکتر از ورودی خوانده می‌شوند. کاراکتری که باید در رشته جست‌وجو شود در CH1 و کاراکتری که باید جایگزین آن شود در CH2 قرار می‌گیرد. هر یک از کاراکترهای رشته ورودی با CH1 مقایسه می‌شوند و چنانچه مساوی باشند، CH2 به‌جای آن قرار می‌گیرد. پس از انجام تغییرات، رشته حاصل به خروجی می‌رود.

```
Datasg      segment         para    'code'
  Msg1      db      'Enter a string: ',"$"
  Msg2      db      'Result string is: ',"$"
  Msg3      db      'Enter source character: ',"$"
  Msg4      db      'Enter target character: ',"$"
  Ch1       db      ?
  Ch2       db      ?
  Strlist   label   byte
  Max       db      20
  Len       db      ?
  Buffer    db      20 dup(' ')
  Dolar     db      '$'
Datasg      ends

Codesg      segment         para    'code'
Mainproc    far
      Assumeds: datasg, cs: codesg
      Mov   ax, datasg
      Mov   ds, ax

      Mov   al, 25          ; تعداد سطر
      Mov   ch, 0
      Mov   cl, 0
      Mov   dh, 24          ; سطر
      Mov   dl, 79          ; ستون
      Mov   bh, 7           ; رنگ
      Mov   ah, 6h
      Int   10h
```

پاک کردن صفحه نمایش

```
        Mov     dh, 10          ; سطر
        Mov     dl, 30          ; ستون
        Mov     bh, 0           ; شماره صفحه
        Mov     ah, 2h
        Int     10h
        Lea     dx, msg1
        Mov     ah, 9h
        Int     21h
        Mov     ah, 0ah
        Lea     dx, strlist
        Int     21h
        Mov     dh, 12          ;سطر
        Mov     dl, 30          ; ستون
        Mov     bh, 0           ; شماره صفحه
        Mov     ah, 2h
        Int     10h
        Lea     dx, msg3
        Mov     ah, 9h
        Int     21h

        Mov     ah, 1h
        Int     21h
        Mov     ch1, al

        Mov     dh, 14          ; سطر
        Mov     dl, 30          ; ستون
        Mov     bh, 0           ; شماره صفحه
        Mov     ah, 2h
        Int     10h

        Lea     dx, msg4
        Mov     ah, 9h
        Int     21h

        Mov     ah, 1h
        Int     21h
        Mov     ch2, al

        Lea     bx, buffer
        Mov     cl, 0
Next:   Mov     dl, [bx]
        Cmp     ch1, dl
        Jne     p2
        Mov     dl, ch2
        Mov     [bx], dl
p2:     Inc     bx
        Inc     cl
        Cmp     cl, len
        Jne     next
```

انتقال مکان نما

چاپ پیام msg1

دریافت رشته

انتقال مکان نما

چاپ پیام msg3

خواندن کاراکتر اول

انتقال مکان نما

چاپ پیام msg4

خواندن کاراکتر دوم

یافتن کاراکتر اول و جایگزینی آن با کاراکتر دوم

```
        Mov     dh, 16
        Mov     dl, 30
        Mov     bh, 0
        Mov     ah, 2h
        Int     10h

        Mov     dx, offset msg2
        Mov     ah, 9h
        Int     21h

        Lea     dx, buffer
        Mov     ah, 9h
        Int     21h

        Mov     ax, 4c00h
        Int     21h
Mainendp
Codesg  ends
        End     main
```

انتقال مکان نما

چاپ msg2

چاپ رشته جدید

مثال از خروجی برنامه:

```
Enter a string: abxymnavfa
Enter source character: a
Enter target character: y
Result string is: ybxymnyvfy
```

۷-۵ انتخاب‌های چندگانه

در برنامه‌هایی که تاکنون نوشته شد، با استفاده از دستور CMP دو مقدار با یکدیگر مقایسه شدند و بر اساس نتیجه مقایسه، دو مسیر مختلف طی شدند و درنتیجه، در هر مسیر، دستورات گوناگونی اجرا شدند. گاهی ممکن است در برنامه‌ای نیاز به چند انتخاب باشد که در آن صورت، به ساختار پیچیده‌تری نیاز هست. در زبان‌های سطح بالا، ساختارهای If-Then-Else و Case (در زبان پاسکال) یا Switch (در زبان C) برای انتخاب‌های چندگانه مورداستفاده قرار می‌گیرند. ازآنجایی‌که با این ساختارها در زبان‌های سطح بالا آشنایی دارید، روش پیاده‌سازی آن‌ها را در زبان اسمبلی بررسی می‌کنیم.

۷-۵-۱ ساختار If – Then – Else

در برنامه‌هایی که تاکنون نوشته شد، به نحوی از این ساختار استفاده‌شده است ولی نامی از این ساختار برده نشد. در این ساختار، ابتدا شرطی تست می‌شود، در صورت درست بودن شرط، مجموعه‌ای از دستورات، وگرنه مجموعه‌ای دیگر از دستورات اجرا می‌شوند. ساختار If به‌صورت زیر است:

```
If    شرط
      <دستورات ۱>
Else
      <دستورات ۲>
End If
```

به‌عنوان‌مثال دستورات زیر را در نظر بگیرید:

```
If    sum < 100     Then
        .                    }
        .                    } دستورات ۱
        .                    }
Else
        .                    }
        .                    } دستورات ۲
        .                    }
End If
```

برای پیاده‌سازی این ساختار در اسمبلی به‌صورت زیر عمل می‌شود:

```
Cmp sum, 100
Jnl  P1
      .                      }
      .                      } دستورات ۱
      .                      }
Jmp End_If

P1:
      .                      }
      .                      } دستورات ۲
      .                      }
End_If: …
```

این ساختار sum را بررسی می‌کند درصورتی‌که sum کوچک‌تر از 100 نبود به P1 پرش می‌کند و دستورات ۲ را اجرا می‌کند. در غیر این صورت دستورات ۱ اجرا می‌شود و برنامه خاتمه می‌یابد.

روش دیگر پیاده‌سازی این ساختار:

```
Cmp sum, 100
Jl   P1
      .                      }
      .                      } دستورات ۲
      .                      }
Jmp End_If

P1 :
      .                      }
      .                      } دستورات ۱
      .                      }
End_If: …
```

این ساختار sum را بررسی می‌کند درصورتی‌که sum کوچک‌تر از 100 بود به P1 پرش می‌کند و دستورات ۱ را اجرا می‌کند. در غیر این صورت دستورات ۲ اجرا می‌شود و برنامه خاتمه می‌یابد.

مثال) مجموعه دستورات زیر را به زبان اسمبلی تبدیل کنید:

```
If ((ax=5) && (cx<8))
{
      DX=DX+4
      BX=BX-1
}
Else
{
      BX=BX+1
      DX=DX+1
}
```

زبان اسمبلی

```
CMP AX, 5
      JNE   ELSE
CMP CX, 8
      JNB    ELSE
ADD   BX, 4
DEC   BX
JMP   END

ELSE:         INC BX
              INC DX
END: …
```

```
If ((ax=5) || (cx<8))
{
      DX=DX+4
      BX=BX-1
}
Else
{
      BX=BX+1
      DX=DX+1
}
```

زبان اسمبلی

```
CMP AX, 5
JE   L1
CMP CX, 8
JB   L1
      INC   BX
      INC   DX
      JMP   END

L1:   ADD DX, 4
      DEC BX
END: …
```

۷-۵-۲ پیاده‌سازی ساختار Switch

ساختار Switch برای تصمیم‌گیری‌های مختلف بر اساس مقدار یک متغیر مورداستفاده قرار می‌گیرد.

ساختار Switch در زبان C به‌صورت زیر پیاده‌سازی می‌شود:

```
Switch      (choice)
{
      Case 1:
            دستورات ۱

      Case 2:
            دستورات ۲

      .
      .
      .
      Default      :
      .
      .
      .
}
```

برای پیاده‌سازی این ساختار با سه مقدار برای choice، در زبان اسمبلی به‌صورت زیر عمل می‌شود:

```
Cmp choice, 1
    Je    L1
Cmp choice, 2
    Je    L2
Cmp choice, 3
    Je    L3
.
.
.
    Jmp   L4
L1 :              ; choice = 1
    Jmp   L5
L2 :              ; choice = 2
    Jmp   L5
L3 :              ; choice = 3
    Jmp   L5
L4 :              ; choice > 1 , 2 , 3
    Jmp   L5
L5 : …
```

مثال) دستور زبان سطح بالای زیر را به زبان اسمبلی تبدیل نمایید:

```
Switch (x)
{
    Case 1:
        CX = CX - 2
    Case 2:
        BX = BX + 1
    Defult
        AX = 10
}
```

تبدیل‌شده به زبان اسمبلی:

```
Cmp X, 1
    JE   CASE1

Cmp X, 2
    JE   CASE2

    JMP Defult

CASE1:
        SUB CX, 2
        JMP END
CASE 2:
        INC BX
        JMP END
Defult:
        Mov   AX, 10
END: …
```

این برنامه بررسی می‌کند اگر X = 1 بود به CASE1 پرش کند، اگر X = 2 بود به CASE2 پرش کند و در غیر این صورت به Defult پرش کند.

۷-۶ حلقه تکرار با دستور Loop

تاکنون با استفاده از دستورات پرشی توانستید برنامه‌های پیچیده‌تری بنویسید. با دستورات پرشی، حلقه‌های تکرار نیز تشکیل دادید و بدین ترتیب، دستوراتی را چندین بار اجرا کردید. استفاده از دستور Loop برای ایجاد حلقه تکرار، ساده‌تر از کاربرد دستورات پرشی برای این کار است. دستور Loop به‌صورت زیر به کار می‌رود:

```
Loop        <برچسب دستور>
```

این دستور ثبات CX را به‌عنوان شمارنده به کار می‌برد لذا قبل از شروع حلقه تعداد دفعات اجرای حلقه می‌بایست در CX ریخته شود. دستور Loop در انتهای حلقه نوشته‌شده و جلوی آن برچسبی می‌آید که ابتدای آن حلقه را مشخص می‌سازد. با اجرای دستور Loop، کنترل اجرای برنامه، به دستوری که برچسب آن در جلوی دستور Loop ذکرشده است، منتقل می‌شود. با هربار اجرای Loop یک واحد از ثبات CX کم می‌شود تا به صفر برسد. وقتی CX صفر شد، حلقه تکرار خاتمه می‌یابد. اعمالی که در حلقه Loop انجام می‌شود عبارت‌اند از:

- از ثبات CX یک واحد کم می‌شود.
- اگر مقدار ثبات CX به صفر برسد، اولین دستور بعد از Loop اجرا می‌شود وگرنه اجرای برنامه از دستوری که برچسب آن در جلوی Loop آمده است اجرا می‌گردد.
- پرش به‌صورت آدرس کوتاه انجام می‌شود (از فاصله ۱۲۸- تا ۱۲۷)

بنابراین فرم کلی این دستور به‌صورت زیر است:

```
Mov CX, تعداد اجرای حلقه
L1:
      .
      .
      .
LOOP L1
```

۷-۷ پیاده‌سازی حلقه‌های تکرار

در زبان‌های برنامه‌سازی سطح بالا، معمولاً دو حلقه تکرار for و while کاربرد زیادی دارند. حلقه for معمولاً برای ایجاد حلقه‌هایی با تعداد دفعات معین و حلقه while برای ایجاد حلقه‌هایی با تعداد دفعات نامعین به کار می‌رود.

۷-۷-۱ نمونه‌هایی از چند حلقه for

۱- حلقه تکراری که دستوران آن ۱۰ بار تکرار می‌شوند:

```
Mov cx, 10
For1:
              .
              .
              .
Loop for1
```

۲- حلقه تکراری که دستورات آن به تعداد دفعاتی که در متغیر number قرار دارد تکرار می‌شوند:

```
Mov cx, number
For2:
            .
            .
            .
Loop For2
```

اگر number برابر صفر باشد، این حلقه ۶۵۵۶۵ بار اجرا می‌شود. علتش این است که پس از یک‌بار اجرای حلقه، مقدار صفر که در ثبات CX قرار دارد به FFFF تبدیل می‌شود (یک واحد از آن کسر می‌گردد) وقتی بار دیگر اجرا می‌شود، مقدار FFFF به FFFE تبدیل می‌شود و این روند آن‌قدر ادامه می‌یابد تا CX به صفر برسد. به همین دلیل، حلقه را به‌صورت زیر بنویسید:

```
Mov cx, number
Jcxz  P1
For2:
            .
            .
            .
Loop For2
P1:  mov    ax, 50
```

در دومین دستور، بررسی می‌شود که اگر محتویات CX صفر است، کنترل اجرای برنامه به P1 می‌رود.

روش دیگر پیاده‌سازی این حلقه به‌صورت زیر است:

```
Mov         cx, number
Cmp         cx, 0
Jle   p1
For1:
            .
            .
            .
Loop for1
P1:  mov    ax, 50
```

۳- همان‌طور که گفته شد، حلقه تکرار Loop با آدرس کوتاه کار می‌کند و لذا در برنامه‌های طولانی، ممکن است به کار نیاید. برای پیاده‌سازی حلقه تکرار در این‌گونه موارد به‌صورت زیر عمل می‌شود (حلقه‌ای با ۱۰ با تکرار):

```
Mov cx, 10
For1:
            .
            .
            .
      Dec   cx
      Jcxz  p1              ; اگر cx صفر بود برو به p1 وگرنه برو به ابتدای حلقه
      Jmp   for1
P1:  mov    ax, 100
```

مثال) برنامه‌ای که رشته و کاراکتری را از ورودی می‌خواند و کاراکتر را در رشته جست‌وجو می‌کند و سپس پیام مناسبی را صادر می‌نماید.

هدف از این برنامه، آشنایی با حلقه تکرار Loop است. طول رشته به‌عنوان شمارنده حلقه تکرار قرار می‌گیرد. (len در cx قرار داده‌شده است.

```
Datasg      segment para 'code'

  Msg1      db      'Enter a string: ',"$"
  Msg2      db      'Enter a character: ',"$"
  Msg3      db      '<< Character exist >>',"$"
  Msg4      db      '<< Character not exist >>',"$"
  Char      db              ?
    شروع لیست پارامتر ;
  Strlist label     byte
  Max       db      20
  Len       db      ?
  Buffer db 20 dup (' ')
  Dolar     db      '$'                    } تعریف رشته

Datasg      ends

Codesg      segment para 'code'
  Main      proc    far
            Assumeds: datasg, cs: codesg

            Mov     ax, datasg
            Mov     ds, ax

            Mov     al, 25
            Mov     ch, 0
            Mov     cl, 0
            Mov     dh, 24
            Mov     dl, 79
            Mov     bh, 7
            Mov     ah, 6h
            Int     10h                    } پاک کردن صفحه نمایش

            Mov     dh, 10
            Mov     dl, 30
            Mov     bh, 0
            Mov     ah, 2h
            Int     10h                    } انتقال مکان نما

            Lea     dx, Msg1
            Mov     ah, 9h
            Int     21h                    } چاپ Msg1

            Mov     ah, 0ah
            Lea     dx, strlist
            Int     21h                    } دریافت رشته
```

```
                Mov     dh, 12
                Mov     dl, 30
                Mov     bh, 0
                Mov     ah, 2h
                Int     10h

                Lea     dx, Msg2
                Mov     ah, 9h
                Int     21h

                Mov     ah, 1h
                Int     21h
                Mov     char, al

                Mov     ah, 2h
                Mov     dh, 14
                Mov     dl, 30
                Mov     bh, 0
                Int     10h

                ; String جستجوی ch در
                Lea     bx, buffer
                Mov     cl, len
Next:           mov     dl, [bx]
                Cmp     char, dl
                Jne     p2
                Lea     dx, msg3
                Mov     ah, 9h
                Int     21h

                Jmp     p3
p2:             Inc     bx
                Loop    next

                Lea     dx, Msg4
                Mov     ah, 9h
                Int     21h

p3:             mov     ax, 4c00h
                Int     21h
Main            endp

Codesg          ends
                End     main
```

انتقال مکان نما

چاپ Msg2

خواندن کاراکتر و انتقال به محل char

انتقال مکان نما

جستجوی کاراکتر در رشته

چاپ Msg4

مثال از خروجی برنامه:

```
Enter a string: alireza
Enter a character: b
<< Character not exist >>
```

۲-۷-۷ پیاده‌سازی حلقه while

حلقه تکرار while در زبان‌های سطح بالا به‌صورت زیر مشخص می‌شود:

```
While (شرط) {
    دستورات
}
```

شرط موجود در حلقه تست می‌شود و چنانچه دارای ارزش درستی باشد، دستورات حلقه تکرار اجرا می‌شوند و این کار آن‌قدر ادامه می‌یابد تا شرط حلقه نقض شود. حلقه زیر را در نظر بگیرید:

```
While (sum < 100) {
    .
    .
    .
}
```

برای پیاده‌سازی این حلقه در زبان اسمبلی به‌صورت زیر عمل می‌شود:

```
While   cmp   sum, 100    ;   sum < 100
        Jnl   EndWhile    ;   jump not less than
        .
        .
        .
        Jmp   while       ;   body of while
EndWhile: …
```

عیب این پیاده‌سازی این است که اگر طول بدنه حلقه بیش از ۱۲۷ بایت باشد، حلقه تکرار به‌درستی عمل نمی‌کند. در این‌گونه موارد از پیاده‌سازی زیر استفاده می‌شود:

```
While:  cmp   sum, 100    ;   sum < 100?
        J1    body
        Jmp   End_while
Body:
        .
        .
        .
        Jmp   while
End_while: …
```

فرض کنید می‌خواهیم لگاریتم عددی مثل NUM ۲ را در پایه دو محاسبه کنیم. لگاریتم پایه دو عدد NUM، بزرگ‌ترین x صحیح است که $num > x^{x}$ باشد. بنابراین، الگوریتم محاسبه لگاریتم مبنای دو عدد را می‌توان به‌صورت زیر نوشت:

```
X = 0;
Sum = 1;
While (sum <= num) {
    sum را در عدد ۲ ضرب کن
    یک واحد به x اضافه کن
}
یک واحد از x کم کن
```

برای پیاده‌سازی این الگوریتم در اسمبلی از دستورات زیر استفاده می‌شود:

```
            Mov     cx, 0           ; log =0 or x=0
            Mov     ax, 1           ; sum = 1
While:      cmp     ax, num
            Jnle    end_while
            Add     ax, ax          ; multiply sum by 2
            Jnc     cx              ; add 1 to x
            Jmp     while
end_while:
            Dec cx          ;       subtract 1 from x
```

معمولاً شرط‌های موجود در حلقه تکرار while ترکیبی از چند شرط ساده‌اند. یعنی دو یا چند شرط با دستور and یا or باهم ترکیب می‌شوند. حلقه تکرار زیر را در نظر بگیرید:

```
While       sum < 100       and     count <= 20
```

با فرض اینکه sum یک کلمه حافظه و مقدار count در ثبات cx قرار دارد برای پیاده‌سازی این حلقه تکرار در اسمبلی به‌صورت زیر عمل می‌شود:

```
While:      cmp     sum, 100            ; sum < 100?
            Jnl     end_while           ; exit if not
            Cmp     cx, 24              ; count c = 24?
            Jnl     end_while           ; exit if not
            …
            Jmp     while
end_while: …
```

اکنون حلقه تکرار زیر را در نظر بگیرید:

```
While       (sum    <100    or      flag=1)
```

با فرض اینکه sum در ثبات ax و flag در ثبات dh قرار دارد، برای پیاده‌سازی آن در اسمبلی به‌صورت زیر عمل می‌شود:

```
While:      cmp     ax, 100         ; sum < 100?
            Jl      body                    ; execute body if so
            Cmp     dh, 1                   ; flag = 1?
            Je      body                    ; execute body if so
            Jmp     end_while               ; exit since nether true
Body:       …
            Jmp     while
end_while
```

۸-۷ دستورات LOOPD

اگر از دستور Loop برای ایجاد حلقه استفاده شود، حلقه تکرار حداکثر 65535 بار تکرار می‌شود. در پردازنده‌های 80386 و بالاتر، دستور LoopD موجب می‌شود تا از ثبات 32 بیتی ECX به‌عنوان شمارنده حلقه تکرار استفاده کرد و درنتیجه حلقه تکرار 2^{32} -1 یعنی 4,294,967,295 بار می‌تواند اجرا شود. به‌عنوان‌مثال، دستورات زیر را در نظر بگیرید:

```
        MOV     ECX, A0000000h
        .
        .
        .
L1:
        .
        .
        .
LoopD   L1      ; Use ecx as loop counter
```

اگر برنامه در حالت 32 بیتی ترجمه شود، دستور Loop، به‌طور خودکار از ثبات ECX به‌عنوان شمارنده استفاده می‌کند. در چنین حالتی برای استفاده از ثبات CX به‌جای ECX، باید از دستور LoopD استفاده کرد.

مثال) با استفاده از ماکروها برنامه‌ای بنویسید که در یک حلقه دائماً عددی را از ورودی دریافت کند و مشخص کند که این عدد زوج است یا فرد؟

```
Include io.h

Sseg segment stack
    DW    32 Dup(?)
Sseg ends

Dseg segment
  STR       DB    4       DUP(?)
  Prompt DB       'enter your number:',0
  Msge      DB    'Even', 13,10,0
  Msgo      DB    'Odd', 13,10,0
  Temp      DB    2
Dseg ends

Cseg segment
Assume cs: cseg, ds: dseg
Start:mov   ax, seg dseg
    Mov   ds, ax
L1: output prompt
    Inputs STR, 4
    Atoi  STR
    Div   temp
    Cmp   ah, 0
    Je    L2
    Jmp   L3
    L2:   output msge
    Jmp   L1
    L3:   output msgo
    Jmp   L1
    Mov   ax, 4c00h
    Int   21h
Cseg ends

    End   start
```

مثال) به کمک ماکروها برنامه‌ای بنویسید که ۷ عدد از کاربر دریافت کرده، ماکزیمم آن‌ها را چاپ کند.

```
Include     io.h
Sseg segment stack
      DW    64      DUP (?)
Sseg ends
Dseg segment
   Value    DB      10      DUP (?)
   Prompt DB        'enter your numbers:', 13,10,0
   Answer DB        13, 10,'The maximom is:',0
   Max      DB      6       DUP (?)
   DB       13,10,0
   Temp     DW      00
Dseg ends
Cseg segment
      Assume cs: cseg, ds: dseg
Start:mov   ax, seg dseg
      Mov   ds, ax
      Output prompt
      Mov   cx, 7
L1:   mov   dx, cx
      Inputs value, 10
      Atoi  value
      Cmp   ax, temp
      JA    L2
L3:   mov   cx, dx
      Loop  L1
      Itoa  max, temp
      Output answer
      Output max
      Jmp   L4
L2:   mov   temp, ax
      Jmp   L3
L4:   mov   ax, 4c00h
      Int   21h
Cseg ends
      End   start
```

فصل هشتم دستورات کار کردن با بیت‌ها

۸-۱ دستورات بولی[1]

در اسمبلی، تعدادی از دستورات وجود دارند که برای کار کردن بابیت‌ها مفیدند. مثلاً می‌توانند بیت‌هایی را به صفر یا یک تبدیل کنند. این دستورات که اغلب دستورات بولی نامیده می‌شوند، عبارت‌اند از AND، OR، XOR، TEST و NOT. این دستورات به‌جز NOT به‌صورت زیر به کار می‌روند:

دستور بولی <عملوند۲> و <عملوند۱>

عملوندهای این دستورات می‌توانند به‌صورت زیر باشند:

عملوند۱	ثبات	حافظه	ثبات	حافظه
عملوند۲	ثبات	ثبات	مقدار ثابت	مقدار ثابت

دستور NOT بر روی یک عملوند عمل می‌کند و به‌صورت زیر به کار می‌رود:

NOT عملوند

عملوند دستور NOT می‌تواند حافظه یا ثبات باشد. در مورد دستورات بولی، به موارد زیر توجه کنید:

- در دستور AND، اگر دو بیت متناظر، یک باشند، نتیجه یک است وگرنه نتیجه صفر است.
- در دستور OR، اگر هر دو بیت متناظر صفر باشند، نتیجه صفر است وگرنه نتیجه یک است.
- در دستور XOR، اگر یکی از بیت‌ها برابر یک و بیت دیگر صفر باشد نتیجه یک است وگرنه نتیجه صفر است. (اگر بیت‌های متناظر، برابر نباشند، نتیجه یک است)
- در دستور TEST مانند دستور AND عمل می‌شود، ولی محتویات عملوند مقصد (عملوند۱) تغییر نمی‌کند.
- در دستور NOT، بیت‌های صفر به یک و بیت‌های یک به صفر تبدیل می‌شوند.

مثال)

$$\begin{array}{r} 0101 \\ \underline{NOT} \\ 1010 \end{array} \qquad \begin{array}{r} 0101 \\ AND \\ \underline{0011} \\ 0001 \end{array} \qquad \begin{array}{r} 0101 \\ OR \\ \underline{0011} \\ 0111 \end{array} \qquad \begin{array}{r} 0101 \\ XOR \\ \underline{0011} \\ 0110 \end{array}$$

برای آشنایی با کاربرد دستورات بولی، به چند مثال توجه کنید. در این مثال‌ها فرض می‌شود که مقدار ثبات BH برابر با 00111010 و مقدار ثبات CH برابر 10100011 است:

AND BL, 0Fh	BL برابر با 00001010 خواهد شد.
AND BL, 00h	BL برابر با 00000000 خواهد شد.
AND BL, Ch	BL برابر با 00100010 خواهد شد.
OR CH, BL	CH برابر با 10111011 خواهد شد.
XOR BL, 0FFh	BL برابر با 11000101 خواهد شد.
XOR BL, BL	BL برابر با 00000000 خواهد شد.

[1] Boolean

```
TEST    DX, OFFh
        JZ          P1 ... اگر DX برابر با صفر بود برو به
TEST    BL, 00000001B
        JNZ         P2 ... اگر BL حاوی عدد فرد بود برو به
OR      DX, DX
        JZ          P3 ... اگر DX برابر با صفر بود برو به
OR      DX, DX
        JS          P4 ... اگر DX منفی بود برو به
```

مثال) برنامه‌ای که عدد موجود در ثبات AX را به رشته تبدیل کرده، در خروجی چاپ می‌کند.

برای چاپ اعداد، باید آن را به رشته تبدیل نمود. بنابراین، تبدیل عدد به رشته، در اسمبلی از اهمیت ویژه‌ای برخوردار است. یکی از نکاتی که باید در تبدیل عدد به رشته به آن توجه داشته باشید این است که، اختلاف هر رقم با کاراکتر به‌اندازه 48 یا 30h است. یعنی برای تبدیل رقم 0 به کاراکتر '0' باید مقدار 30h را به صفر اضافه کرد. نکته دیگر در تبدیل عدد به رشته این است که، عدد موردنظر بر 10 تقسیم می‌شود و باقیمانده آن، رقم سمت راست، عددی است که در ثبات DL قرار می‌گیرد. باقیمانده را با 30h جمع کرده، نتیجه را در رشته‌ای به طول 6 و به نام string قرار می‌دهیم. کاراکترهای به‌دست‌آمده، از آخرین محل به‌طرف اولین محل رشته string قرار داده می‌شوند و سپس علامت عدد که قبلاً تعیین شد و در متغیر sign قرار گرفت، به ابتدای رشته اضافه می‌شود. اگر عدد مثبت باشد کاراکتر ' ' وگرنه علامت '-' در ابتدای رشته قرار می‌گیرد.

```
Stksg       segment         stack

    Db      64      dup ("stack")

Stksg       ends

Datasg      segment para 'code'

  Msg       db      'Numstring is: ',"$"
  Sign      db      ?
  String    db      6 dup (' '), "$"

Datasg      ends

Codesg      segment para 'code'

  Main      proc    far

            Assumeds: datasg, cs: codesg

            Mov     ax, datasg
            Mov     ds, ax

            Lea     bx, string
            Add     bx, 5
            Mov     ax, -2456       ; این مقدار باید به رشته تبدیل شود
```

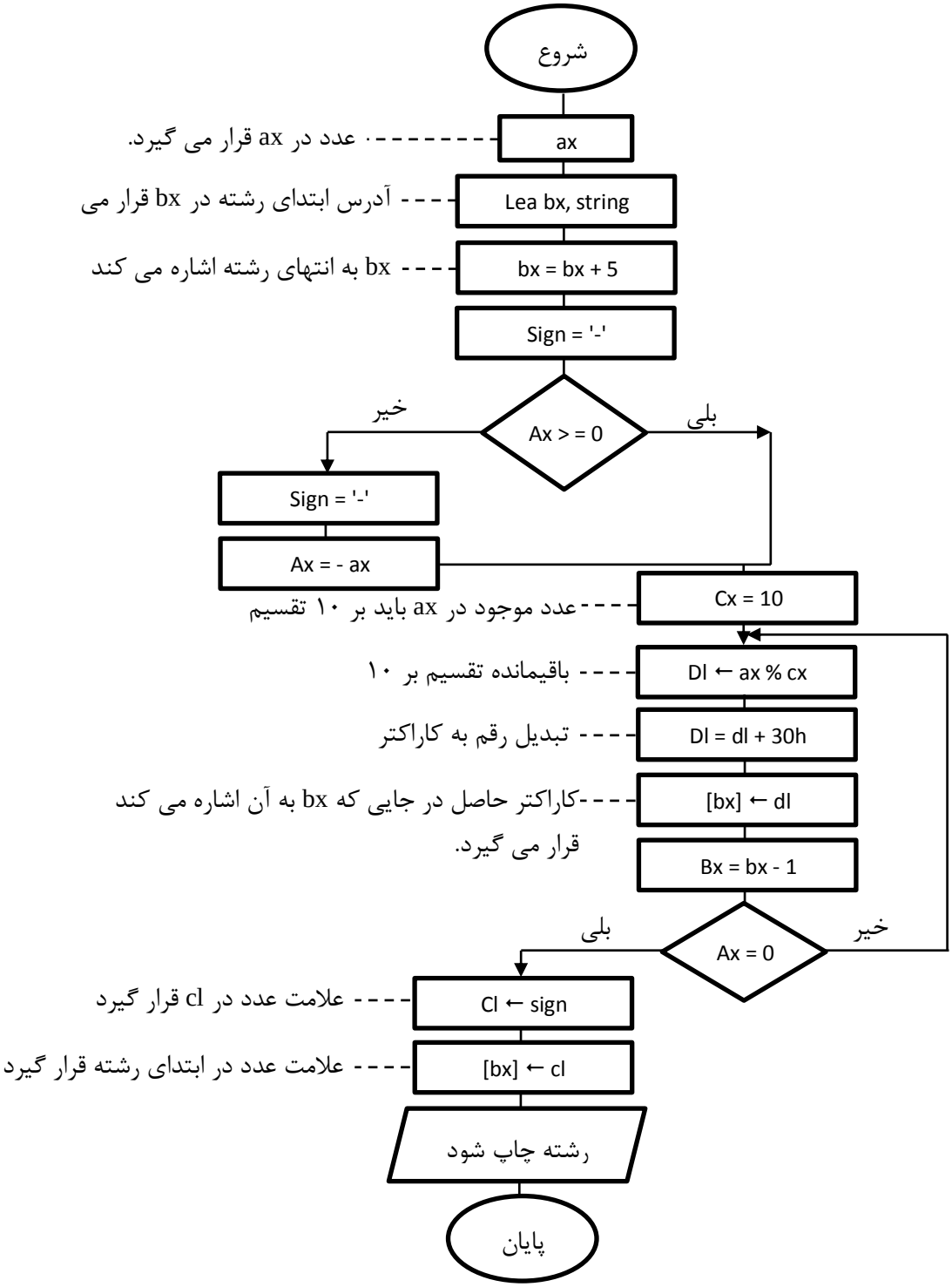
```
            Mov     sign, ' '       ; علامت عدد
            Cmp     ax, 0
            Jge     setup           ; اگر عدد منفی بود برو به setup
            Mov     sign, '-'       ; وگرنه علامت - را در sign قرار بده
            Neg     ax              ; و علامت عدد را جهت محاسبات حذف کن

Setup:      mov     cx, 10          ; divisor
Divloop:    mov     dx, 0           ; گسترش شماره به double word
            Div     cx              ; تقسیم کردن به ۱۰
            Add     dl, 30h         ; تبدیل باقیمانده از عدد به کاراکتر
            Mov     [bx], dl        ; قرار دادن کاراکتر در رشته
            Dec     bx
            Cmp     ax, 0
            Jne     divloop
            Mov     cl, sign        ; الحاق علامت عدد به رشته
            Mov     [bx], cl

            Mov     al, 25          ┐
            Mov     ch, 0           │
            Mov     cl, 0           │
            Mov     dh, 24          │
            Mov     dl, 79          ├ پاک کردن صفحه نمایش
            Mov     bh, 7           │
            Mov     ah, 6h          │
            Int     10h             ┘

            Mov     dh, 10          ┐
            Mov     dl, 30          │
            Mov     bh, 0           ├ انتقال مکان نما
            Mov     ah, 2h          │
            Int     10h             ┘

            Lea     dx, Msg         ┐
            Mov     ah, 9h          ├ چاپ Msg
            Int     21h             ┘

            Lea     dx, string      ┐
            Mov     ah, 9h          ├ چاپ عدد رشته ای
            Int     21h             ┘

            Mov     ax, 4c00h
            Int     21h

  Main      endp

Codesg      ends

            End     main
```

مثال از خروجی برنامه:

با توجه به این‌که در داخل برنامه، عدد 2456- در ثبات AX قرار داده شد، پس از تبدیل به رشته، به‌صورت زیر چاپ‌شده است:

Numstring is = -2456

فلوچارت تبدیل عدد دودویی به رشته جهت چاپ برنامه قبل به شکل زیر است:



مثال) برنامه‌ای که حروف بزرگ رشته‌ای را به حروف کوچک تبدیل می‌کند.

تبدیل حروف بزرگ به کوچک و یا بالعکس، در موارد متعددی به کار می‌آید. مقادیر اسکی حروف بزرگ A تا Z عبارت‌اند از 41h تا 5Ah و مقادیر اسکی حروف کوچک a تا z برابر است با 61h تا 7Ah. نمایش حروف A و a در زیر مشخص‌شده است:

	۷	۶	۵	۴	۳	۲	۱	۰
A:	1	0	0	0	0	0	0	1

	۷	۶	۵	۴	۳	۲	۱	۰
a:	1	0	1	0	0	0	0	1

همان‌طور که می‌بینید، اختلاف a و A در بیت شماره ۵ است. یعنی بیت شماره ۵ حرف a برابر با یک و بیت شماره ۵ حرف A برابر با صفر است. این قاعده در مورد تمام حروف کوچک و بزرگ صادق است. بنابراین برای تبدیل حرف کوچک به حرف بزرگ، کافی است بیت شماره ۵ برابر با صفر شود. برای این کار کافی است حرف بزرگ را با مقدار بیت 00100000b، XOR کنیم. برای تشخیص این‌که حرفی از حروف بزرگ است یا نه، باید توجه داشته باشید که حروف بزرگ بین 41h و 5Ah و حروف کوچک بین 61h تا 7Ah هستند.

```
Datasg    segment     para    'code'
  msg1    db      'Enter a capital letter string: ',"$"
  msg2    db      'the result string is: ',"$"
  Strlist label   byte
  Max     db      20
  Len     db      ?
  Buffer  db      20 dup(' ')
  Dolar   db      '$'
Datasg    ends
Codesg    segment     para    'code'
Mainproc  far
          Assumeds: datasg, cs: codesg
          Mov     ax, datasg
          Mov     ds, ax

          Mov     al, 25
          Mov     ch, 0
          Mov     cl, 0
          Mov     dh, 24
          Mov     dl, 79
          Mov     bh, 7
          Mov     ah, 6h
          Int     10h

          Mov     dh,  10
          Mov     dl, 30
          Mov     bh, 0
          Mov     ah, 2h
          Int     10h

          Lea     dx, msg1
          Mov     ah, 9h
          Int     21h
```

پاک کردن صفحه نمایش

انتقال مکان نما

چاپ msg1

```
        Mov     ah, 0ah
        Lea     dx, strlist      } دریافت رشته
        Int     21h

        Lea     bx, buffer              ; آدرس رشته در bx
        Mov     cl, len                 ; طول رشته در cl
        Mov     ch, 0
Next:   mov     ah, [bx]
        Cmp     ah, 41h         ; آیا کاراکتر بین 41h و 5Ah است؟
        Jb      p1
        Cmp     ah, 5ah
        JA      p1
        Xor     ah, 00100000b   ; بله، تبدیل کردن به حرف کوچک
        Mov     [bx], ah
p1:     Inc     bx
        Loop    next

        Mov     dh, 12          } انتقال مکان نما
        Mov     dl, 30
        Mov     bh, 0
        Mov     ah, 2h
        Int     10h

        Lea     dx, msg2        } چاپ msg2
        Mov     ah, 9h
        Int     21h

        Lea     dx, buffer      } چاپ خروجی
        Mov     ah, 9h
        Int     21h

        Mov     ax, 4c00h
        Int     21h
Main    endp

Codesg  ends
    End main
```

مثال از خروجی برنامه:

```
Enter a capital letter string: MOHAMMAD
The result string is: Mohammad
```

مثال) برنامه‌ای که رشته عددی را از ورودی می‌خواند و به عدد دودویی تبدیل می‌کند و در ثباتی ذخیره می‌نماید.

اطلاعات عددی که از ورودی خوانده می‌شوند به‌صورت رشته‌ای ذخیره می‌گردند که برای انجام محاسبات بر روی آن‌ها، باید به مقادیر عددی تبدیل شوند. لذا تبدیل کدهای اسکی به مقادیر عددی، در اسمبلی از اهمیت ویژه‌ای برخوردار است. الگوریتم آن به‌صورت زیر است:

```
Value = 0 (مقدار عددی)
While          (تا زمانی که ارقام تمام نشده‌اند)
{
      Value = Value * 10
      کد کاراکتر را به رقم صحیح تبدیل کن
      این مقدار را به Value اضافه کن
      به کاراکتر بعدی برو
}
End  while
```

روش انجام کار به این صورت است: ابتدا فاصله‌های خالی ابتدای رشته عددی حذف می‌شود و سپس علامت عدد تشخیص داده می‌شود. باوجود علامت منها در رشته ورودی، مقدار 1- وگرنه مقدار یک در متغیر sign قرار می‌گیرد. پس از تشخیص علامت عدد، باید مشخص شود که آیا کاراکتر بعدی بین '0' تا '9' هست یا خیر. اگر نباشد، کاراکتر مجاز نیست و عمل تبدیل کد اسکی به رقم صحیح، خاتمه می‌یابد. ولی اگر کاراکتر بین '0' تا '9' باشد، به رقم 0 تا 9 تبدیل می‌شود. برای تبدیل کد اسکی کاراکتر به رقم، کد اسکی، با مقدار 000fh عمل AND صورت می‌گیرد. با این کار، تمام بیت‌های عدد به‌جز چهار بیت سمت راست، صفر می‌شود و چهار بیت سمت راست، تعیین‌کننده مقدار صحیح است. مثلاً کد اسکی ۳۷ که مربوط به عدد ۷ است با AND شدن با مقدار 000fh به 0007 تبدیل می‌شود. مقادیر حاصل در ثبات AX قرار می‌گیرند. (مقدار اولیه ثبات AX صفر است). برای اضافه کردن رقم جدید به مقدار قبلی AX، ابتدا به مقدار AX در ۱۰ ضرب می‌شود و سپس رقم جدید به محتویات جدید AX اضافه می‌شود.

پس از تبدیل رشته عددی به عدد و قرار گرفتن در ثبات AX، مقدار موجود در ثبات AX به رشته تبدیل‌شده، در صفحه‌نمایش ظاهر می‌گردد تا از نتیجه کار آگاهی پیدا کنید. توجه داشته باشید که حالت سرریزی در این برنامه تست نشده است. حالت سرریز وقتی است که عدد حاصل بین 32767 و 32768- نباشد.

```
Sseg segment stack
   DW    64 DUP (?)
Sseg ends
Dseg segment

      Msg1   db      'Enter a number: ',"$"
Msg2         db      'The result string number is: ',"$"
Count        db      0
Sign DW      ?
String       db      6 dup (' '), "$"
Sign1        db      ?

Strlist      label   byte
Max db       20
Len  db      ?
Buffer       db      20 dup (' ')
Dolar        db      '$'

Dseg ends
```

```
Cseg segment
Assume cs: cseg, ds: dseg
Start:          Mov    ax, seg dseg
                Mov    ds, ax
                Mov    al, 25
                Mov    ch, 0
                Mov    cl, 0
                Mov    dh, 24
                Mov    dl, 79
                Mov    bh, 7
                Mov    ah, 6h
                Int    10h
                Mov    dh, 10
                Mov    dl, 30
                Mov    bh, 0
                Mov    ah, 2h
                Int    10h
                Lea    dx, msg1
                Mov    ah, 9h
                Int    21h
                Mov    ah, 0ah
                Lea    dx, strlist
                Int    21h
                Lea    bx, buffer              ; آدرس رشته در bx
while_blank:    cmp    byte ptr [bx], ' '      ; حذف کردن خالی
                Jne    end_while_blank         ; while خروج از
                Inc    bx
                Jmp    while_blank
end_while_blank: Mov   sign, 1                 ; پیش‌فرض علامت مثبت
                Cmp    byte ptr [bx], '+'      ; + جستجو
                Je     skip_sign               ; اگر
                Cmp    byte ptr [bx], '-'      ; جستجو
                Jne    save_sign
                Mov    sign, -1
Skip_sign:      Inc    bx
Save_sign:      mov    ax, 0
                Mov    count, 0
While_digit:    cmp    byte ptr [bx], '0'
                Jl     end_while_digit
                Cmp    byte ptr [bx], '9'
                Jg     end_while_digit
                Mov    cx, 10
                Mul    cx
                Mov    cl, [bx]
                And    cx, 000fh
                Add    ax, cx
                Inc    count
                Inc    bx
                Jmp    while_digit
End_while_digit:
```

پاک کردن صفحه نمایش

انتقال مکان نما

چاپ msg1

دریافت رشته

```
                    Imul    sign
    ; //////////////////////      تبدیل عدد موجود در ax به رشته
                    Lea     bx, string
                    Add     bx, 5
                    Mov     sign, ' '
                    Cmp     ax, 0
                    Jge     setup           ; اگر عدد منفی نبود برو به setup
                    Mov     sign1, '-'      ; وگرنه علامت منها را در sign1 قرار بده
                    Neg     ax
Setup:              Mov     cx, 10
Divloop:            Mov     dx, 0
                    Div     cx
                    Add     dl, 30h
                    Mov     [bx], dl
                    Dec     bx
                    Cmp     ax, 0
                    Jne     divloop
                    Mov     cl, sign1
                    Mov     [bx], cl
    ; //////////////////////

                    Mov     dh, 12
                    Mov     dl, 30
                    Mov     bh, 0
                            Mov     ah, 2h
                    Int     10h

                    Lea     dx, msg2
                    Mov     ah, 9h
                    Int     21h

                    Lea     dx, string
                    Mov     ah, 9h
                    Int     21h

                    Mov     ax, 4c00h
                    Int     21h
Cseg        ends
        End     start
```

انتقال مکان نما

چاپ msg2

چاپ خروجی

مثال از خروجی برنامه:

```
Enter a number: -1245
The result string number is: -1245
```

مثال) برنامه‌ای که لگاریتم پایه ۲ یک عدد مثبت را تعیین می‌کند. لگاریتم پایه ۲ عدد، بزرگ‌ترین مقدار صحیح x است، به‌طوری‌که:

عدد موردنظر => 2x

عدد موردنظر به‌صورت رشته خوانده می‌شود و پس از تبدیل به عدد، در ثبات AX قرار می‌گیرد. سپس بر اساس الگوریتم محاسبه لگاریتم، مقدار X محاسبه‌شده در ثبات CX قرار داده می‌شود. مقدار ثبات CX به ثبات AX منتقل می‌گردد تا پس از تبدیل‌شدن به رشته، در خروجی چاپ شود.

```
Sseg segment stack
    DW    64 DUP (?)
Sseg ends
Dseg segment
    Msg1      db      'Enter a number: ',"$"
    Msg2      db      'the log of this number is: ',"$"
    Count     db      0
    Sign      DW      ?
    String    db      6 dup (' '), "$"
    Sign1     db      ?
    Number    DW      ?
    Strlist   label   byte
    Max       db      20
    Len       db      ?
    Buffer    db      20 dup (' ')
    Dolar     db      '$'
Dseg ends
Cseg segment
        Assume cs: cseg, ds: dseg
        Start:          Mov   ax, seg dseg
                        Mov   ds, ax
                        Mov    al, 25
                        Mov    ch, 0
                        Mov    cl, 0
                        Mov    dh, 24
                        Mov    dl, 79
                        Mov    bh, 7
                        Mov    ah, 6h
                        Int    10h                  } پاک کردن صفحه نمایش

                        Mov    dh, 10
                        Mov    dl, 30
                        Mov    bh, 0
                        Mov    ah, 2h
                        Int    10h                  } انتقال مکان نما

                        Lea    dx, msg1
                        Mov    ah, 9h
                        Int    21h                  } چاپ msg1

                        Mov    ah, 0ah
                        Lea    dx, strlist
                        Int    21h                  } دریافت رشته

                        Lea    bx, buffer           ; آدرس رشته در bx
    while_blank:        cmp    byte ptr [bx], ' '   ; حذف کردن خالی
                        Jne    end_while_blank      ; خروج از while
```

```
                    Inc     bx
                    Jmp     while_blank
end_while_blank:    Mov     sign, 1             ; پیش‌فرض علامت مثبت
                    Cmp     byte ptr [bx], '+'
                    Je      skip_sign
                    Cmp     byte ptr [bx], '-'
                    Jne     save_sign
                    Mov     sign, -1
Skip_sign:          Inc     bx
Save_sign:          mov     ax, 0
                    Mov     count, 0
While_digit:  cmp   byte ptr [bx], '0'
                    Jl      end_while_digit
                    Cmp     byte ptr [bx], '9'
                    Jg      end_while_digit
                    Mov     cx, 10
                    Mul     cx
                    Mov     cl, [bx]
                    And     cx, 000fh
                    Add     ax, cx
                    Inc     count
                    Inc     bx
                    Jmp     while_digit
End_while_digit:    Imul    sign
      ; //////////////////////    محاسبه لگاریتم
                    Mov     number, ax
                    Mov     cx, 0
                    Mov     ax, 1
Lopwhile:           cmp     ax, number
                    Jnle    end_while
                    Add     ax, ax
                    Inc     cx
                    Jmp     lopwhile
End_while:          dec     cx
                    Mov     ax, cx
; //////////////////////    تبدیل عدد موجود در ax به رشته
                    Lea     bx, string
                    Add     bx, 5
                    Mov     sign, ' '
                    Cmp     ax, 0
                    Jge     setup               ; اگر عدد منفی نبود برو به setup
                    Mov     sign1, '-'          ; وگرنه علامت منها را در sign1 قرار بده
                    Neg     ax
Setup:              Mov     cx, 10
Divloop:            Mov     dx, 0
                    Div     cx
                    Add     dl, 30h
                    Mov     [bx], dl
                    Dec     bx
                    Cmp     ax, 0
                    Jne     divloop
```

```
                Mov     cl, sign1
                Mov     [bx], cl

                Mov     dh, 12
                Mov     dl, 30
                Mov     bh, 0
                        Mov     ah, 2h
                Int     10h

                Lea     dx, msg2
                Mov     ah, 9h
                Int     21h

                Lea     dx, string
                Mov     ah, 9h
                Int     21h

                Mov     ax, 4c00h
                Int     21h
Cseg    ends
        End     start
```

انتقال مکان نما

چاپ msg2

چاپ خروجی

مثال از خروجی برنامه:

```
Enter a number: 32
The log of this number is: 5
```

مثال) برنامه‌ای که کاربرد ساده‌ای از دستور loop را جهت حلقه سازی نشان می‌دهد. در این برنامه، ابتدای ثبات‌های al و bl برابر با صفر می‌شوند و در یک حلقه تکرار ۵ تایی، هربار یک واحد به این ثبات‌ها اضافه می‌شود و سپس این دو ثبات باهم جمع شده، نتیجه در ثبات ax قرار می‌گیرد. عدد موجود در ثبات ax به کد اسکی تبدیل‌شده، در خروجی چاپ می‌شود.

```
Datasg      segment para 'code'
  Msg       db      'this result is: ',"$"
  Sign      db              ?
  String    db      6 dup (' '), "$"
Datasg      ends
Codesg      segment para 'code'
  Main      proc    far
            Assumeds: datasg, cs: codesg
            Mov     ax, datasg
            Mov     ds, ax

            Mov     ah, 0
            Mov     al, 0
            Mov     bl, 0
            Mov     cx, 5
            P1:     Inc     al
            Inc     bl
            Loop    p1
            Add     al, bl
            Lea     bx, string
            Add     bx, 5
            Mov     sign, ' '
```

```
            Cmp     ax, 0
            Jge     setup
Setup:      mov     cx, 10
Divloop:    mov     dx, 0
            Div     cx
            Add     dl, 30h
            Mov     [bx], dl
            Dec     bx
            Cmp     ax, 0
            Jne     divloop
            Mov     cl, sign
            Mov     [bx], cl
            Mov     ah, 6h

            Mov     al, 25
            Mov     ch, 0
            Mov     cl, 0
            Mov     dh, 24
            Mov     dl, 79
            Mov     bh, 7
            Mov     ah, 6h
            Int     10h

            Mov     dh, 10
            Mov     dl, 30
            Mov     bh, 0
            Mov     ah, 2h
            Int     10h

            Lea     dx, Msg
            Mov     ah, 9h
            Int     21h

            Lea     dx, string
            Mov     ah, 9h
            Int     21h

            Mov     ax, 4c00h
            Int     21h
  Main      endp
Codesg      ends
            End     main
```

پاک کردن صفحه نمایش

انتقال مکان نما

چاپ Msg

چاپ خروجی

مثال از خروجی برنامه:

```
The result is: 10
```

۸-۲ شیفت دادن

با استفاده از دستورالعمل‌های منطقی (بولی) که در بخش قبل به آن‌ها پرداخته شد، می‌توان بیت‌های موجود در یک ثبات یا بایت‌ها و کلمات حافظه را دست‌کاری کرد. با دستورالعمل‌های شیفت می‌توان موقعیت بیت‌های کلمه یا بایت را تغییر داد. شیفت بر دو نوع است: شیفت منطقی و شیفت حسابی. شیفت منطقی را شیفت بدون علامت و شیفت حسابی را شیفت با علامت نیز می‌گویند.

بیت‌ها را می‌توان به سمت راست یا به سمت چپ شیفت داد. برای شیفت منطقی به راست، از دستور SHR و برای شیفت حسابی به راست از دستور SAR استفاده می‌شود. دستور SHL برای شیفت منطقی به چپ و دستور SAL برای شیفت حسابی به چپ مورداستفاده قرار می‌گیرد. نحوه کاربرد دستورات شیفت به‌صورت زیر است:

<دستورات شیفت> <عملوند ۱>, <عملوند ۲>

عملوند ۱، ثبات یا حافظه‌ای است که بیت‌های آن باید شیفت پیدا کنند و عملوند ۲ تعداد بیت‌هایی را که باید شیفت پیدا کنند مشخص می‌کند. عملوند ۲ می‌تواند یک مقدار عددی یا ثبات CL باشد. در پردازنده‌های8086 و 8088 مقدار عددی در عملوند ۲، فقط می‌تواند مقدار یک باشد و برای انجام شیفت‌های بیشتر، باید تعداد موردنظر را در ثبات CL قرارداد، ولی در پردازنده‌های 80286 و بالاتر، مقدار ثابت عددی می‌تواند تا ۳۱ باشد.

۸-۲-۱ شیفت به راست

در شیفت منطقی به راست، هر بیت که از سمت راست عملوند ۱ خارج می‌شود، یک بیت صفر از سمت چپ وارد می‌شود تا بیت‌های خالی را پر کند ولی در شیفت حسابی به راست، با خارج شدن هر بیت از سمت راست، برای پر کردن بیت‌های خالی، از بیت علامت استفاده می‌شود. بنابراین عملکرد SHR و SAR باهم متفاوت است. توجه به این نکته نیز مهم است که هر بیت که در اثر شیفت از عملوند اول خارج می‌شود، در فلگ رقم نقلی (CF) قرار می‌گیرد. نمونه‌هایی از عملکرد SHR در جدول زیر آمده است.

CF	دهدهی	عملکرد	توضیح	دستورالعمل
-	183	10110111	مقداردهی به bh	Mov bh, 10110111b
1	91	01011011	شیفت یک بیت به راست	Shr bh, 1
			تعیین تعداد شیفت	Mov ch, 2
1	22	00010110	شیفت ۲ بیت به راست	Shr bh, cl
1	5	00000101	شیفت ۲ بیت به راست	Shr bh, 2

در آخرین دستور مقدار ۲ برای تعیین تعداد شیفت به‌کاررفته است و این دستور در کامپیوترهای 80286 و بالاتر قابل‌اجرا است. نمونه‌هایی از کاربرد دستورالعمل SAR در جدول زیر آمده است.

CF	دهدهی	عملکرد	توضیح	دستورالعمل
-	-73	10110111	مقداردهی به bh	Mov bh, 10110111b
1	-37	11011011	شیفت یک بیت به راست	Sar bh, 1
			تعیین تعداد شیفت	Mov ch, 2
1	-10	11110110	شیفت ۲ بیت به راست	Sar bh, cl
1	-3	11111101	شیفت ۲ بیت به راست	Sar bh, 2

هر شیفت به راست، معادل تقسیم کردن عدد بر ۲ است و شیفت‌های ۲ بیتی معادل تقسیم کردن بر ۴ است. تقسیم نیز به‌صورت صحیح انجام می‌شود. یعنی حاصل تقسیم اعدادی مثل ۵ بر ۲ برابر با ۲ است. پس از شیفت دادن، می‌توان با دستور JC[1] (پرش در صورت وجود رقم نقلی) بیت شیفت داده‌شده به فلگ رقم نقلی را تشخیص داد (صفر یا یک).

۸-۲-۲ شیفت به چپ

همان‌طور که گفته شد، برای شیفت منطقی به چپ از دستور SHL و برای شیفت حسابی به چپ از دستور SAL استفاده می‌شود. بیتی که در دستورات شیفت به چپ، از سمت چپ خارج می‌شود، در فلگ رقم نقلی (CF) قرار می‌گیرد. هر تعداد از بیت‌هایی که از سمت چپ خارج می‌شوند. به همان تعداد، صفر از سمت راست وارد می‌شوند تا بیت‌های خالی را پر کنند، لذا دو دستور SHL و SAL کاملاً یکسان عمل می‌کنند و تفاوتی بین آن‌ها وجود ندارد. نمونه‌هایی از کاربرد دستورات شیفت به چپ، در جدول زیر آمده است. همان‌طور که در جدول می‌بینید، هر عمل شیفت به چپ معادل ضرب کردن در عدد ۲ است و شیفت ۲ بیتی معادل ضرب کردن در ۴ است. استفاده از عمل شیفت برای ضرب، سریع‌تر از دستورالعمل‌های مربوط به ضرب است.

دستورالعمل	توضیح	عملکرد	ده‌دهی	CF
Mov bh, 10110101b	مقداردهی به bh	00000101	5	-
Sal bh, 1	شیفت یک بیت به چپ	00001010	10	**0**
Mov ch, 2	تعیین تعداد شیفت			
Sal bh, cl	شیفت ۲ بیت به چپ	00101000	40	**0**
Sal bh, 2	شیفت ۲ بیت به چپ	10100000	160	**0**

۸-۳ دوران بیت‌ها

دستورالعمل‌های دوران بیت‌ها، خیلی شبیه به دستورالعمل‌های شیفت هستند. دستورالعمل‌های شیفت، بیت‌هایی که شیفت داده می‌شوند، از ثبات حافظه خارج می‌شوند و از طرف دیگر، صفر یا علامت عدد (در شیفت حسابی به راست) وارد می‌شوند تا بیت‌های خالی را پر کنند. ولی در دستورالعمل‌های دوران، بیت‌هایی که ازیک‌طرف به خارج از ثبات یا حافظه منتقل می‌شوند، از طرف دیگر واردشده، فضای خالی را پر می‌کنند و ضمناً در فلگ رقم نقلی نیز قرار می‌گیرند. دوران نیز مانند شیفت به دو صورت انجام می‌شود:

- دوران منطقی (بدون علامت)
- دوران حسابی (با علامت)

دستورالعمل‌های ROR و RCR برای دوران به راست و دستورالعمل‌های ROL و RCL برای دوران به چپ مورداستفاده قرار می‌گیرند. این دستورات به‌صورت زیر به کار می‌روند:

<دستورات دوران> <عملوند ۱>, <عملوند ۲>

[1] Jump if Carry

عملوند ۱ ثبات یا حافظه‌ای است که باید دوران پیدا کند و عملوند ۲، مقدار ثبات یا ثبات cl است که تعداد دوران را مشخص می‌کند. در پردازنده‌های 8086 و 8088 اگر تعداد دوران بیش از یک باشد، تعداد باید در ثبات CL قرار گیرد.

۸-۳-۱ دوران بیت‌ها به راست

دوران ROR برای دوران منطقی به راست و دستور RCR برای دوران حسابی به راست مورداستفاده قرار می‌گیرد. در دستور ROR هر بیت که دوران داده می‌شود در فلگ CF نیز قرار می‌گیرد و در دستور RCR هر بیت که باید دوران یابد ابتدا در CF و سپس از CF از سمت چپ وارد می‌شود. نمونه‌ای از کاربرد دستور ROR در جدول زیر آمده است.

CF	عملکرد	توضیح	دستورالعمل
-	10110111	مقداردهی به bh	Mov bh, 10110111b
1	11011011	دور آن‌یک بیت به راست	Ror bh, 1
		تعیین تعداد دوران	Mov ch, 3
0	01111011	دوران ۳ بیت به راست	Ror bh, cl
0	01101111	دوران ۳ بیت به راست	Ror bh, 3

۸-۳-۲ دوران بیت‌ها به چپ

برای دوران بیت‌ها به چپ از دستورالعمل‌های ROL و RCL استفاده می‌شود. دستور ROL برای دوران منطقی و دستور RCL برای دوران حسابی به کار می‌رود. در دستورالعمل ROL ابتدا بیت دوران یافته از سمت راست وارد می‌شود و سپس در فلگ CF قرار می‌گیرد ولی در دستور RCL بیتی که دوران می‌یابد ابتدا به فلگ CF می‌رود و سپس محتویات CF از سمت راست ثبات یا حافظه وارد می‌شود. نمونه‌هایی از کاربرد دستورات دوران چپ در جدول زیر آمده است.

CF	عملکرد	توضیح	دستورالعمل
-	10110111	مقداردهی به bh	Mov bh, 10110111b
1	01101111	شیفت یک بیت به چپ	Rol bh, 1
		تعیین تعداد شیفت	Mov ch, 2
1	01111011	شیفت ۲ بیت به چپ	Rol bh, cl
1	11011011	شیفت ۲ بیت به چپ	Rol bh, 2

مثال)

برنامه‌ای که کد اسکی عددی را از ورودی خوانده، پس از تبدیل آن به عدد دودویی، در ثبات ax قرار می‌دهد. سپس عدد موجود در ثبات ax را به‌عنوان چهار رقم مبنای شانزده نمایش می‌دهد. برای این منظور، چهار گروه

چهار بیتی باید از مقدار موجود در ثبات ax جدا شوند. هر گروه چهار بیتی نشان‌دهنده یک مقدار ده‌دهی از صفرتا ۱۵ است و هر گروه برای نمایش باید به کاراکتر تبدیل شود. این کاراکتر، رقمی بین صفرتا ۹ برای مقادیر (0000) تا (1001) یا کاراکترهای A تا F برای مقادیر (1010) تا (1111) است. الگوریتم این برنامه را می‌توان به‌صورت زیر طراحی کرد:

```
For   count = 4 to 1 loop
   کپی کن     dx را به ax
   تمام بیت‌ها به‌غیراز چهار بیت آخر را صفر کن
   If value in dx <= 9 Then
      مقدار موجود در dx را به یک کاراکتر '0' تا '9' تبدیل کن
      مقدار موجود در dx را به یک کاراکتر 'A' تا 'F' تبدیل کن
   End if
      کاراکتر را در آدرسی از حافظه که با bx مشخص می‌شود ذخیره کن
      از bx یک واحد کم کن تا به محل بعدی سمت چپ اشاره کند
      مقدار موجود در ثبات ax را چهار بیت به‌طرف راست شیفت بده
End  for
```

دو دستور در این برنامه ضروری به نظر می‌رسد: دستور or dx, 30h مثل دستور add dx, 30h است که قبلاً نیز آن را در تبدیل عدد به رشته مشاهده کردید. این دستور موجب می‌شود تا کد عدد به کد اسکی تبدیل شود. دستور دیگر، دستور add dx, 'A' - 10 است. کد 'A' برابر با ۶۵ است که پس از کسر شدن ۱۰، به ۵۵ تبدیل می‌شود لذا این دستور معادل Add dx, 55 است. کار این دستور این است که رقم موجود در ثبات AX را به کاراکترهای 'A' تا 'F' تبدیل کند. اگر این دستور را به‌صورت Add dx, 'a' - 10 به کار ببرید حروف کوچک a تا f را خواهید داشت.

```
Sseg segment stack
    DW    64 DUP (?)
Sseg ends

Dseg segment
    Msg1      db      'Enter a number: ',"$"
    Msg2      db      'the result string number is: ',"$"
    Count     db      0
    Sign      DW      ?
    String    db      4 dup (' '), "$"
    Sign1     db      ?
    Strlist   label   byte
    Max       db      20
    Len       db      ?
    Buffer    db      20 dup (' ')
    Dolar     db      '$'
Dseg ends
Cseg segment
        Assume cs: cseg, ds: dseg
        Start:      Mov   ax, seg dseg
                    Mov   ds, ax
```

```
                Mov     al, 25
                Mov     ch, 0
                Mov     cl, 0
                Mov     dh, 24
                Mov     dl, 79
                Mov     bh, 7
                Mov     ah, 6h
                Int     10h
                Mov     dh, 10
                Mov     dl, 30
                Mov     bh, 0
                Mov     ah, 2h
                Int     10h
                Lea     dx, msg1
                Mov     ah, 9h
                Int     21h
                Mov     ah, 0ah
                Lea     dx, strlist
                Int     21h
                Lea     bx, buffer              ; آدرس رشته در bx
while_blank:    cmp     byte ptr [bx], ' '      ; حذف کردن خالی
                Jne     end_while_blank         ; خروج از while
                Inc     bx
                Jmp     while_blank
end_while_blank: Mov    sign, 1                 ; پیش‌فرض علامت مثبت
                Cmp     byte ptr [bx], '+'
                Je      skip_sign
                Cmp     byte ptr [bx], '-'
                Jne     save_sign
                Mov     sign, -1
Skip_sign:      Inc     bx
Save_sign:      mov     ax, 0
                Mov     count, 0
While_digit:    cmp     byte ptr [bx], '0'
                Jl      end_while_digit
                Cmp     byte ptr [bx], '9'
                Jg      end_while_digit
                Mov     cx, 10
                Mul     cx
                Mov     cl, [bx]
                And     cx, 000fh
                Add     ax, cx
                Inc     count
                Inc     bx
                Jmp     while_digit
End_while_digit: Imul   sign
                Lea     bx, string + 3
                Mov     cx, 4
Forloop:        mov     dx, ax
```

پاک کردن صفحه نمایش

انتقال مکان نما

چاپ msg1

دریافت رشته

```
                And     dx, 000fh
If_1:           Cmp     dx, 9
                Jnle    else_1
Then_1:         or      dx, 30h
                Jmp     short endif_1
Else_1:         add     dx, 'A' - 10
Endif_1:        Mov     byte ptr [bx], dl
                Dec     bx
                Shr     ax, 4
                Loop    forloop
                Mov     dh, 12
                Mov     dl, 30
                Mov     bh, 0
                Mov     ah, 2h
                Int     10h
                Lea     dx, msg2
                Mov     ah, 9h
                Int     21h
                Lea     dx, string
                Mov     ah, 9h
                Int     21h

                Mov     ax, 4c00h
                Int     21h
Cseg    ends
        End     start
```

انتقال مکان نما

چاپ msg2

چاپ خروجی

فصل نهم زیر برنامه‌ها

مسائلی که تاکنون حل کرده‌ایم ساده و کوچک بودند. برای حل برنامه پیچیده، باید آن را به بخش‌های کوچک‌تری تقسیم کرد، به‌طوری‌که هر بخش کار خاصی را انجام دهد و برای هر بخش نیز برنامه خاصی را نوشت. برنامه‌ای که برای حل بخشی از مسئله نوشته می‌شود **زیر برنامه** نام دارد. تقسیم کردن برنامه بزرگ به زیر برنامه، دارای مزایایی است که برخی از آن‌ها عبارت‌اند از:

- خوانایی برنامه را بالا می‌برد، زیرا اهداف کلی برنامه را می‌توان در برنامه اصلی مشاهده کرد.
- استفاده از زیر برنامه، کارگروهی را امکان‌پذیر می‌سازد.
- از برنامه نوشته‌شده دیگران می‌توان استفاده نمود.
- از زیر برنامه نوشته‌شده در برنامه‌های مختلف می‌توان به‌دفعات استفاده کرد. این کار دو مزیت دارد:
  - حجم برنامه کم می‌گردد.
  - اگر زیر برنامه یک‌بار نوشته شود و تست گردد هرگاه در برنامه دیگری مورداستفاده قرار گیرد، مطمئن هستیم این زیر برنامه خطایی ندارد.
- رفع اشکال زیر برنامه‌ها آسان‌تر است. زیرا یک برنامه بدون استفاده از زیر برنامه، طولانی شده این امر ممکن است اشکالاتی به وجود آورد.
- می‌توان برنامه‌های پرکاربرد را نوشت و کتابخانه‌ای از این برنامه‌ها را ایجاد کرد.
- فقط با استفاده از این روش به‌سادگی می‌توان با زبان‌های برنامه‌سازی دیگر، ارتباط برقرار کرد.

سرعت طراحی برنامه‌ها بالا می‌رود.

۹-۱ جنبه‌های مختلف زیر برنامه

هر زیر برنامه دو جنبه دارد:

۱- جنبه تعریف

۲- جنبه فراخوانی

جنبه تعریف زیر برنامه، دستورالعمل‌هایی است که عملکرد زیر برنامه را مشخص می‌کند و **جنبه فراخوانی**، دستوری است که آن را اجرا می‌کند. برای اجرای هر زیر برنامه، باید آن را فراخوانی کرد. برنامه‌ای که زیر برنامه را فراخوانی می‌کند، **برنامه فراخوان** و زیر برنامه‌ای که فراخوانی می‌شود، زیر برنامه **فراخوانده شده** نام دارد.

فرم کلی برنامه اسمبلی با زیر برنامه‌ها:

```
   Stksg   segment              ; سگمنت پشته
   .
   .
   Stksg ends
; -----------------------------------
   Datasg segment               ; سگمنت داده
   .
   .
   Datasg   ends
; -----------------------------------
```

```
   Eseg segment                ; سگمنت اضافی
   .
   .
   Eseg   ends
; ------------------------------------
   Codsg    segment            ; سگمنت کد
   Main   proc    far
   .                           ; برنامه اصلی
   .
   Main    endp
; ------------------------------------
   Proc1     proc              ; زیر برنامه ۱
   .
   .
   Proc1     endp
; ------------------------------------
   Proc2     proc         ; زیر برنامه ۲
   .
   .
   Proc2      endp
; ------------------------------------
   Procn    proc          ; زیر برنامه n
   .
   .
   Procn    endp
; ------------------------------------
   Codsg    ends
   End      main
```

۹-۲ انواع زیر برنامه

در اسمبلی، دو نوع زیر برنامه وجود دارد که عبارت‌اند از: زیر برنامه داخلی و زیر برنامه خارجی. زیر برنامه داخلی زیر برنامه‌ای است که در همان فایل برنامه وجود دارد. زیر برنامه خارجی، زیر برنامه‌ای است که در فایل مجزا از فایل برنامه اصلی، وجود دارد.

همان‌طور که مشاهده شد در برنامه اسمبلی، سگمنت پشته و پس‌ازآن سگمنت داده، سگمنت اضافی و سگمنت کد تعریف می‌شوند. سگمنت کد از چند زیر برنامه تشکیل می‌شود. زیر برنامه‌ها بعد از برنامه اصلی تعریف می‌شوند.

۹-۳ تعریف زیر برنامه

زیر برنامه با استفاده از راهنمای proc و endp تعریف می‌شود. چون این دو کلمه، راهنما می‌باشند، دستورالعمل‌های ماشین را تولید نمی‌کنند و راهنمایی برای اسمبلر هستند تا ابتدا و انتهای زیر برنامه مشخص گردد.

هر زیر برنامه با proc شروع و با endp خاتمه می‌یابد:

```
نام زیر برنامه   proc    نوع زیر برنامه
                 بدنه زیر برنامه
                 Ret / retf
نام زیر برنامه   endp
```

هر زیر برنامه دارای نامی است که برای مراجعه به آن به کار می‌رود. نام زیر برنامه از قوانین نام‌گذاری شناسه‌ها (متغیر) تبعیت می‌کند. بعد از نام زیر برنامه راهنمای proc و سپس نوع زیر برنامه قرار می‌گیرد.

نوع زیر برنامه می‌تواند near یا far باشد. درصورتی‌که نوع زیر برنامه far باشد، آن را می‌توان توسط برنامه‌های دیگر فراخوانی کرد. این نوع زیر برنامه، با برنامه اصلی در یک سگمنت قرار دارد.

بدنه زیر برنامه، مجموعه دستوراتی است که عملکرد زیر برنامه را مشخص می‌کند. دستورات ret/retf برای برگشت به برنامه فراخوان به کار می‌رود که در ادامه تشریح می‌شوند. خاتمه زیر برنامه بانام زیر برنامه و شبه دستور endp مشخص می‌گردد. نام قبل از شبه دستور endp باید همان نامی باشد که قبل از دستور proc قرارگرفته است. تعریف زیر را در نظر بگیرید:

```
Proc1    proc far
         بدنه زیر برنامه
         Ret/ retf
Proc1    endp
```

در این مثال نام زیر برنامه proc1 در نظر گرفته‌شده است و نوع آن far است. مفهوم far این است که زیر برنامه را می‌توان توسط هر برنامه دیگری فراخوانی کرد، چه آن برنامه مربوط به این سگمنت باشد، یا بیرون از این سگمنت قرار داشته باشد. زیر برنامه فوق با دستور endp خاتمه یافت.

چون سیستم‌عامل برنامه را فراخوانی می‌کند، لذا برنامه اصلی باید از نوع far باشد، زیرا dos در سگمنت دیگری قرار دارد. مجموعه دستورات زیر، زیر برنامه get را از نوع near تعریف می‌کند:

```
Get   proc
      .
      .
      .
            Ret / retf
Get   endp
```

۹-۴ فراخوانی زیر برنامه

برای فراخوانی زیر برنامه از دستور Call استفاده می‌شود. این دستور به‌صورت زیر به کار می‌رود:

```
Call آدرس
```

آدرس را می‌توان یک آدرس مستقیم در نظر گرفت که معمولاً در فراخوانی زیر برنامه‌ها آدرس مستقیم است. در این روش، آدرس، همان نام زیر برنامه است که بعد از دستور Call قرار می‌گیرد. اگر آدرس غیرمستقیم باشد

در این صورت، آدرس، کلمه‌ای از حافظه است که حاوی آدرس شروع زیر برنامه است. به‌عنوان‌مثال دستور زیر، زیر برنامه get را فراخوانی می‌کند:

```
Call  get
```

نمونه‌ای از فراخوانی زیر برنامه از طریق آدرس غیرمستقیم در زیر آمده است. در این مثال، فرض شده که آدرس زیر برنامه، در حافظه‌ای که آدرس آن با list + 5 مشخص می‌شود قرار دارد:

```
Call    list [5]
```

هنگامی‌که دستور Call اجرا می‌گردد دو عمل انجام می‌شود:

۱- آدرس برگشت به زیر برنامه فراخوان، در پشته ذخیره می‌گردد. آدرس برگشت، آدرس دستورالعمل بعد از Call است. اگر نوع زیر برنامه فراخوانی شده NEAR باشد دو بایت آدرس آفست (IP)، در پشته ذخیره می‌گردد وگرنه چهار بایت آدرس فیزیکی، ابتدا آدرس موجود در ثبات CS و سپس آفست در پشته ذخیره می‌گردد.

۲- کنترل اجرای برنامه به آدرس شروع زیر برنامه می‌رود. به‌عنوان‌مثال، دستورات زیر را در نظر بگیرید:

```
Call  proc1
Mov bx, 1000h
```

در این دستورات، زیر برنامه proc1 فراخوانی می‌شود و پس از اجرای زیر برنامه، کنترل اجرای برنامه از زیر برنامه به برنامه اصلی برمی‌گردد و آدرس برگشت که همان آدرس دستورالعمل MOV است، از پشته بازیابی می‌گردد.

نکته:

اگر زیر برنامه از نوع FAR باشد باید آدرس فیزیکی به اسمبلر داده شود. یعنی چهار بایت آدرس سگمنت و آدرس آفست، و اگر نوع زیر برنامه NEAR باشد باید دو بایت آدرس آفست به اسمبلر داده شود.

۹-۵ نکاتی در نوشتن زیر برنامه‌ها

۱- در ابتدای هر زیر برنامه باید ثبات‌ها را ذخیره کرد و در انتهای زیر برنامه بازیابی نمود، لذا در ابتدای زیر برنامه دستور PUSHA و در انتهای آن دستور POPA قرار می‌گیرد.

۲- هر زیر برنامه باید توضیحات کافی داشته باشد، به‌طوری‌که نام زیر برنامه، هدف، ورودی موردنیاز، خروجی که تولید می‌کند و زیر برنامه‌هایی را که فراخوانی می‌کند مشخص نماید.

۳- در انتهای هر زیر برنامه باید دستور RET / RETF قرار گیرد تا کنترل اجرا، به برنامه فراخوان برگردد.

مثال) برنامه‌ای که با استفاده از زیر برنامه cls صفحه‌نمایش را پاک می‌کند و توسط زیر برنامه move_cursor مکان‌نما را به وسط صفحه‌نمایش انتقال می‌دهد و با استفاده از زیر برنامه disp_message پیام مناسبی را نمایش می‌دهد. هدف از این برنامه آشنایی با تعریف زیر برنامه و فراخوانی آن است.

برنامه example: در این برنامه ابتدا آدرس برگشت به DOS در پشته ذخیره می‌گردد و پس‌ازآن زیر برنامه‌های cls، mov_cursor و disp_message به ترتیب فراخوانی می‌شوند.

زیر برنامه cls: این زیر برنامه برای پاک کردن صفحه‌نمایش به کار می‌رود که از تابع 06h، وقفه 10h استفاده می‌کند.

زیر برنامه mov_cursor: این زیر برنامه مکان‌نما را به سطر ۱۲ ستون ۳۰ انتقال می‌دهد که از تابع 02h، وقفه 10h استفاده می‌نماید.

زیر برنامه disp_message: این زیر برنامه برای نمایش پیامی که در آدرس شناسه mesg1 قرار دارد به کار می‌رود که از تابع 09h وقفه 21h استفاده می‌کند. در انتهای پیام باید کاراکتر "$" وجود داشته باشد.

```
.286c
Sseg segment          'stack'
      Db 128 dup("?")
Sseg ends
Dseg segment
      Mesg1  db       "test program with procdure","$"
Dseg ends

Cseg segment          'code'
Assume      cs: cseg, ss: sseg, ds: dseg
Example     proc    far
      Push    ds                ; data segment ذخیره
      Push    0                 ; ذخیره آدرس بازگشت
      Mov     ax, dseg
      Mov     ds, ax            ; ds = desg
      Call    cls
      Call    move_cursor
      Call    disp_message
      Ret
Example     ENDP
;*************************************
;     Clear   the     screen
;*************************************
Cls   proc    near
      Mov     ah, 6h
      Mov     al, 25
      Mov     ch, 0
      Mov     cl, 0
      Mov     dh, 24
      Mov     dl, 79
      Mov     bh, 61
      Int     10h
      Ret
Cls   endp
;*************************************
;     Move    cursor
;*************************************
Move_cursor proc    near
      Mov     ah, 2h
      Mov     dh, 12
      Mov     dl, 30
      Mov     bh, 0
      Int     10h
      Ret
Move_cursor endp
```

```
;************************************
;       Print     Prompt
;************************************
Disp_message          proc      near
        Lea       dx, mesg1
        Mov       ah, 9h
        Int       21h
        Ret
Disp_message          endp
Cseg ends
        End       example
```

خروجی:

در وسط صفحه، پیام test program with procedure را نمایش می‌دهد.

نکات:

۱- دستورات PUSHA ,POPA تنها در پردازنده‌های ۲۸۶ به بعد قابل‌استفاده‌اند لذا به هنگام استفاده از این دستورات در اول برنامه حتماً باید دستور c۲۸۶. نوشته شود.

۲- بجای استفاده از وقفه خاتمه برنامه در پایان برنامه اصلی، می‌توان از دستور Ret برای بازگشت به فراخوان کننده برنامه اصلی(سیستم‌عامل dos) استفاده کرد ولی در این صورت باید حتماً دو دستور PUSH ds و PUSH 0 را نیز به اول تابع اضافه کرد. (علت آن این است که آدرس برگشت پس از فراخوانی برنامه‌ها توسط dos، DS:0000 است که باید در ابتدای کار توسط برنامه‌نویس در پشته ذخیره شود).

۹-۶ انتقال پارامترها

در فراخوانی زیر برنامه‌ها گاهی لازم است اطلاعاتی بین برنامه اصلی و زیر برنامه‌ها مبادله شوند. این اطلاعات را **پارامتر** گویند. پارامترها بر دو نوع‌اند:

۱- پارامترهای ورودی

۲- پارامترهای خروجی

پارامترهای ورودی اطلاعاتی هستند که برنامه فراخوان به زیر برنامه می‌فرستند و پارامترهای خروجی آن‌هایی هستند که از زیر برنامه فراخوانی شده، برگردانده می‌شوند. دو روش انتقال پارامتر وجود دارد که عبارت‌اند از:

۱- انتقال پارامترها از طریق ثبات‌ها
۲- انتقال پارامترها از طریق پشته

۹-۶-۱ انتقال پارامترها با استفاده از ثبات‌ها

اگر تعداد پارامترها کم باشد می‌توان پارامترها را با استفاده از ثبات‌ها منتقل کرد. مجموعه دستورات زیر با استفاده از ثبات DX، آدرس شناسه prompt را که محتویات آن باید روی صفحه‌نمایش، نشان داده شود، به زیر برنامه disp_message منتقل می‌کند.

```
Lea   dx, prompt
Call  disp_message
```

مثال) برنامه‌ای که با استفاده از زیر برنامه write_char حروف A تا Z را نمایش دهد. هدف از این برنامه آشنایی با زیر برنامه و روش انتقال پارامتر با استفاده از ثبات است.

در ابتدای این برنامه آدرس برگشت به DOS ذخیره‌شده، سپس کاراکتر 'A' در ثبات DL قرار می‌گیرد و با استفاده از یک حلقه تکرار که تعداد تکرار در ثبات CX قرار دارد، زیر برنامه write_char برای نمایش کاراکتر تولیدشده فراخوانی می‌شود و به محتویات ثبات DL، برای تولید کاراکتر بعدی یک واحد اضافه می‌گردد. برای نمایش کاراکتر از تابع 02h وقفه 21h استفاده‌شده است. در این تابع، کاراکتری که چاپ می‌شود در ثبات DL قرار دارد. در این زیر برنامه انتقال پارامتر با استفاده از ثبات DL انجام‌شده است.

```
.286c
Sseg segment        para    public 'stack'
      Db      64 dup('?')
Sseg ends
Cseg segment        para    public  'code'
Assume      cs: cseg, ss: sseg
Example     proc far
      Push    ds
      Push    0
      Mov     dl, 'A'
      Mov     cx, 26
loop1:      call    write_char
            Inc     dl
            Loop    loop1
            Ret
Example     endp
;************************************
;     Display a       char
;************************************
Write_char  proc    near
      Mov     ah, 02
      Int     21h
      Ret
Write_char  endp
Cseg ends
      End     example
```

خروجی:

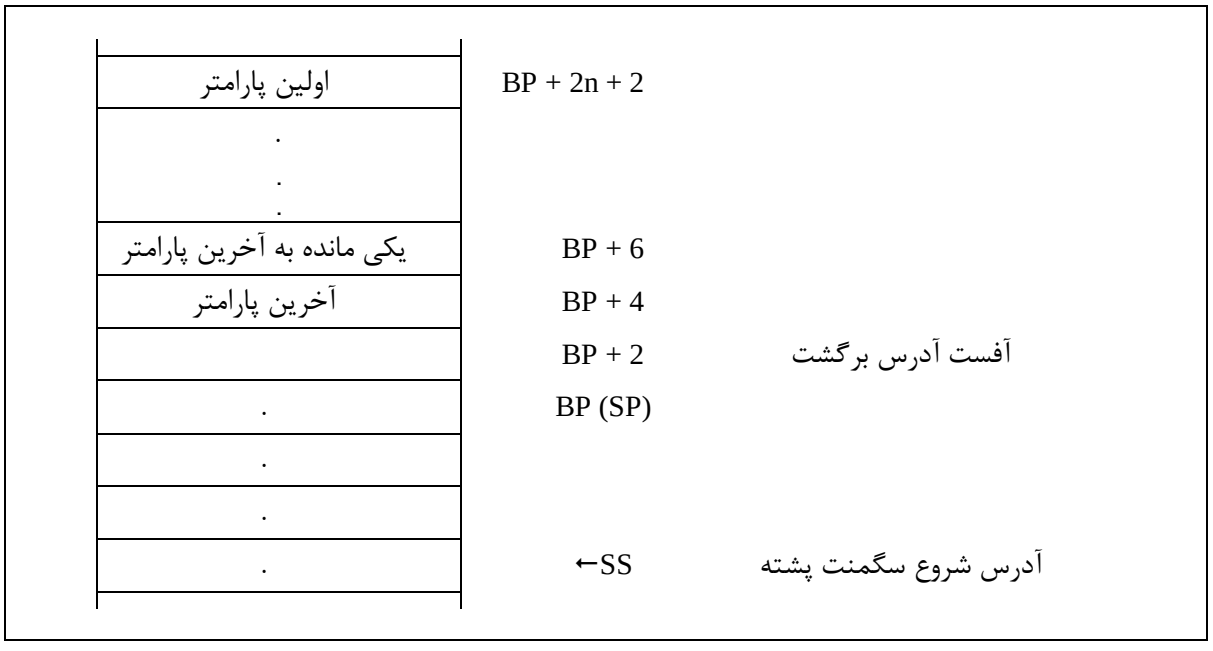
```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

نکته

توجه داشته باشید که پارامترهای خروجی را نیز می‌توان با استفاده از ثبات‌ها به برنامه فراخوان برگرداند. این عمل باید بعد از دستور POPA باشد، چون اگر قبل از این دستور قرار گیرد، مقادیر خروجی که در ثبات قرارگرفته‌اند با دستور POPA از بین می‌روند و مقادیر داخل پشته جایگزین مقادیر خروجی می‌شوند.

۹-۶-۲ انتقال پارامترهای ورودی توسط پشته

هنگامی‌که تعداد پارامترها زیاد باشد برای انتقال پارامترها نمی‌توان از ثبات‌ها استفاده نمود، زیرا تعداد ثبات‌ها محدود است. در این موارد، برای انتقال پارامترها باید از پشته استفاده نمود. بار انتقال پارامترها از طریق پته، قبل از فراخوانی زیر برنامه، پارامترها را در پشته قرار می‌دهیم. چون تمام پارامترها قبل از دستور Call در پشته قرار می‌گیرند، پس از اجرای دستور Call آدرس برگشت به برنامه فراخوان در بالای پشته قرار خواهد گرفت. لذا قبل از دستیابی به پارامترها باید آدرس برگشت را از بالای پشته برداریم. یک روش انجام این کار این است که آدرس برگشت به برنامه فراخوان را در مکانی ذخیره کنیم و هنگام برگشت از زیر برنامه، آن را در بالای پشته قرار دهیم. این روش، دستور Call را شبیه‌سازی می‌نماید که نیازی به این کار نیست. روش دیگر این است که در ابتدای زیر برنامه، محتویات ثبات SP را در ثبات BP قرار دهیم، یعنی در ابتدای زیر برنامه، دستور MOV BP, SP را داشته باشیم. با این کار، در زیر برنامه‌های near، پارامترها از آخرین پارامتر به اولین پارامتر به ترتیب در آدرس‌های آفست [BP+4]، [BP+6]، [BP+8]، [BP+10] و ... قرار می‌گیرند. شکل زیر نحوه قرار گرفتن پارامترها را در پشته، در زیر برنامه‌های نوع near نشان می‌دهد (n تعداد پارامترها را مشخص می‌کند). اگر تعداد پارامترها را سه در نظر بگیریم، پارامتر اول در [BP+8]، پارامتر دوم در [BP+6] و پارامتر سوم در [BP+4] قرار می‌گیرد. اگر مقدار ۳ در را در فرمول BP+2n+2 قرار دهیم آدرس پارامتر اول BP+8 به دست می‌آید.



شکل زیر فراخوانی یک زیر برنامه نوع near را با پارامترهای P1، P2 و P3 و نحوه قرار گرفتن پارامترها را در پشته نشان می‌دهد.

در زیر برنامه نوع far پارامترها از آخرین پارامتر به اولین پارامتر به ترتیب در آدرس‌های [BP+6]، [BP+8]، [BP+10] و ... قرار می‌گیرند. شکل زیر نحوه قرار گرفتن پارامترها را در پشته ، در زیر برنامه‌های نوع far نشان می‌دهد. (n نشان‌دهنده تعداد پارامترهاست):

شکل زیر فراخوانی یک زیر برنامه far با پارامترهای a1، a2، a3 و a4 را نشان می‌دهد:

نکته:

برای برگرداندن پارامترهای خروجی به برنامه فراخوان، می‌توان از پشته استفاده کرد. این پارامترها را نمی‌توان در زیر برنامه توسط دستور PUSH در پشته قرارداد. چون با استفاده از دستور POPA که در انتهای زیر برنامه قرار می‌گیرد، تمام این مقادیر در ثبات‌ها بازیابی می‌گردند. در این صورت نه‌تنها مقادیر ثبات‌ها را از دست می‌دهیم بلکه پارامترها هم به‌عنوان خروجی، به برنامه فراخوان برگردانده نمی‌شوند. برای حل این مشکل، از ثبات BP به‌عنوان اشاره‌گری برای دسترسی و تصحیح پارامترها استفاده می‌شود. درنتیجه پس از بازگشت به برنامه فراخوان، توسط POP می‌توان این پارامترها را برگرداند. البته باید توجه داشت که این پارامترها برعکس ورود به پشته، بازیابی شوند. پس اولین پارامتری که بازیابی می‌نماییم، آخرین پارامتری است که وارد پشته شده و به همین ترتیب پارامترهای دیگر را بازیابی می‌نماییم.

مثال) برنامه‌ای که سه متغیر p1، p2 و p3 را با مقادیر 1111h و 2222h و 3333h تعریف نموده سپس توسط تابع add3 جمع کرده و حاصل را چاپ کند.

```
.286c
Include io.h
Sseg segment stack
      DW     64      Dup(?)
Sseg ends

Dseg segment
      P1     DW     1111h
      P2     DW     2222h
      P3     DW     3333h
      S      DW     ?
Dseg ends
```

```
Cseg segment
Mainproc     far
Assume cs: cseg, ds: dseg
        Mov     ax, seg dseg
        Mov     ds, ax
        Push    P1
        Push    P2
        Push    P3
        Call    add3
        Mov     ax, 4c00h
        Int     21h
Main            endp

Add3            proc    near
        Mov     bp, sp
        Push    ax
        Push    bx
        Push    cx
        Mov     ax, [bp+6]
        Mov     bx, [bp+4]
        Mov     cx, [bp+2]
        Add     ax, bx
        Add     ax, cx
        Mov     S, ax
        Pop     cx
        Pop     bx
        Pop     ax
        Mov ax, s
        Itoa s, ax
        Output s
        Ret
Add3            endp

Cseg ends
        End     main
```

خروجی:

```
26214
```

مثال) برنامه‌ای که به کمک ماکروها عدد N را از ورودی دریافت کرده به تابعی ارسال کند و در آن تابع جمع اعداد از ۱ تا N را محاسبه و به برنامه اصلی برگرداند و این حاصل جمع در برنامه اصلی چاپ شود.

```
.286c
Include         io.h
Sseg segment stack
        DW      64      Dup(?)
Sseg ends

Dseg segment
        Prompt DB       'enter your number:' 0
        Msg     DB      'adding numbers 1 ... N is:' 0
```

```
        Value   DW      10      DUP(?)
        Num     DW       ?
        Sum     DW      ?
        K       DB      2
Dseg ends
Cseg segment
        Assume cs: cseg, ds: dseg
Mainproc        far
        Mov     ax, seg dseg
        Mov     ds, ax
        Output prompt
        Inputs value, 10
        Atoi    value
        Mov     num, ax
        Call    ADDNUM
        Itoa    value, sum
        Output msg
        Output value
        Mov     ax, 4c00h
        Int     21h
Mainendp
ADDNUM  proc    near
        PushA
        Mov     ax, num
        Add     ax, 1
        Mul     num
        Div     K
        Mov     sum, ax
        PopA
        Ret
ADDNUM  endp
Cseg ends
        End     main
```

خروجی:

```
Enter your number:4
adding numbers 1 ... N is:    10
```

مثال) برنامه‌ای که به کمک ماکروها یک ماتریس ۵*۳ را تعریف کرده، سطر اول را با عدد ۱، سطر دوم را با عدد ۲ و سطر سوم را با عدد ۳ پُر کند. آنگاه ابتدا سطر سوم، سپس سطر دوم و در آخر سطر اول را در ۳ خط مجزا چاپ کند.

```
.286c
Include io.h
Sseg segment stack
        DW      64      DUP(?)
Sseg ends
Dseg segment
        Table   DW      15 DUP(?)
        Num     DB      6 DUP (' '), 0
        Blank   DB      13,10,0
Dseg ends
```

```
Cseg segment
        Assume cs: cseg, ds: dseg
Main:   mov     ax, seg dseg
        Mov     ds, ax
        Mov     bx, 0
        Mov     si, 0
        Mov     cx, 5
L1:     mov     table [bx][si],1
        Push    table [bx][si]
        Inc     si
        Loop    L1
        Mov     bx, 1
        Mov     si, 0
        Mov     cx, 5
L2:     mov     table [bx][si],2
        Push    table [bx][si]
        Inc     si
        Loop    L2
        Mov     bx, 2
        Mov     si, 0
        Mov     cx, 5
L3:     mov     table [bx][si],3
        Push    table [bx][si]
        Inc     si
        Loop    L3
        Mov     cx, 5
L4:     pop     ax
        Itoa    num, ax
        Output num
        Loop    L4
        Output blank
        Mov     cx, 5
L5:     pop     ax
        Itoa    num, ax
        Output num
        Loop    L5
        Output blank
        Mov     cx, 5
L6:     pop     ax
        Itoa    num, ax
        Output num
        Loop    L6
        Output blank

        Mov     ax, 4c00h
        Int     21h
Cseg ends
        End     main
```

خروجی:

```
3       3       3       3       3
2       2       2       2       2
1       1       1       1       1
```

فصل دهم ارتباط زبان‌های سطح بالا با اسمبلی

یکی از نکات جالب زبان اسمبلی این است که می‌توان با زبان‌های سطح بالا (مثل پاسکال و C) ارتباط برقرار کرد. این ارتباط به دو صورت امکان‌پذیر است. درروش اول می‌توان در هر مکانی از برنامه زبان‌های C و پاسکال، از یک یا چند دستور اسمبلی استفاده کرد که این روش دارای محدودیت‌های زیر است:

۱- محدودیت استفاده از آدرس‌ها و عملوندها
۲- فقدان تنظیم اولیه مقادیر و متغیرها
۳- نیاز به ذخیره ثبات‌ها
۴- کاهش قابل‌حمل بودن برنامه
۵- کم بودن سرعت ترجمه
۶- محدودیت بهینه‌سازی
۷- محدودیت اشکال‌زدایی

درروش دوم می‌توان زیر برنامه اسمبلی را که در فایل جداگانه‌ای قرار دارد در این زبان‌ها، فراخوانی کرد. در این فصل ارتباط زبان اسمبلی با دو زبان سطح بالای C و پاسکال موردبررسی قرار می‌گیرد.

۱۰-۱ ارتباط زبان اسمبلی با پاسکال

۱۰-۱-۱ دستورات اسمبلی در برنامه پاسکال

برای نوشتن دستورات اسمبلی در برنامه پاسکال، از دستور asm به‌صورت زیر استفاده می‌شود:

```
Asm
    دستورات اسمبلی
end
```

اگرچند دستور اسمبلی در یک سطر باشند، هر دستور باید به ; ختم شود وگرنه نیاز به ; نیست.

مثال) برنامه‌ای که با استفاده از دستورات اسمبلی در برنامه پاسکال، مکان‌نما را به نقطه‌ای از صفحه‌نمایش منتقل کرده کاراکتری را ۱۰ بار در آنجا تایپ می‌کند و سپس کاراکتری با علامت قلب را ۵ بار به‌صورت چشمک‌زن نمایش می‌دهد.

```
Program test;
    Uses dos, CRT;
    Var regs: registers; {For Windows: Tregisters}
Begin
 Clrscr;
 Asm
    Mov   ah, 02        {function}
    Mov   bh, 0         {page#}
    Mov   dh, 10        {row}
    Mov   dl, 50        {col}
    Int   10h           {call bios}
    Mov   ah, 09        {function}
    Mov   al, '!'       {Char to print}
    Mov   bh, 0         {page#}
```

```
        Mov     bl, 7           {attribute}
        Mov     cx, 10          {number of time to print char}
        Int     10h             {call bios}
        Mov     ah, 09          {request diaplay}
        Mov     al, 03h  {picture of heart}
        Mov     bh, 0           {page#}
        Mov     bl, 0f0h        {blink}
        Mov     cx, 05          {five time print heart}
        Int     10h             {call bios}
    End;
end.
```

۱۰-۱-۲ استفاده از زیر برنامه‌های اسمبلی

برای استفاده از زیر برنامه‌های اسمبلی در پاسکال، باید آن زیر برنامه را در فایل جداگانه‌ای تایپ کنید و استفاده از مترجم (tasm یا masm) آن را به فایل obj تبدیل نمایید و در پاسکال نیز آن زیر برنامه را به‌صورت external تعریف کنید. برای لینک کردن زیر برنامه اسمبلی به پاسکال، از راهنمای کامپایلر پاسکال $L به‌صورت زیر استفاده نمایید:

```
$L    <نام فایل obj اسمبلی>
```

فایل obj، زیر برنامه‌ای به زبان اسمبلی است که با این دستور به برنامه پاسکال پیوند زده می‌شود.

در برنامه اسمبلی، بایستی نام زیر برنامه، خارجی تعریف‌شده در پاسکال، به‌صورت public تعریف شود تا لینکر این دو برنامه را به هم پیوند دهد. برای تبدیل فایل اسمبلی به obj از مترجم tasm یا masm به‌صورت زیر استفاده می‌شود:

```
TASM        نام زیر برنامه اسمبلی
        یا
MASM        نام زیر برنامه اسمبلی
```

مثال) برنامه‌ای که با استفاده از زیر برنامه اسمبلی به نام cls، صفحه‌نمایش را پاک می‌کند.

برنامه پاسکال

```
Program test;
        Uses dos;
Procedure    cls;      far;      external;          {$L      cls.obj}
Begin
        Cls;
        Write ('new cursor location!');
end.
```

برنامه cls اسمبلی

```
Cseg segment    para public 'code'
        Public      cls
        Assume      cs: cseg
Cls   proc      far
```

```
        Mov     ax, 6h
        Mov     bh, 7
        Mov     cx, 0
        Mov     dx, 184fh
        Int     10h
        Retf
Cls     endp
Cseg    ends
        End
```

۱۰-۲ انتقال پارامترها از پاسکال به اسمبلی

برای انتقال پارامترها از زیر برنامه پاسکال به برنامه اسمبلی، می‌توان به دو روش عمل کرد. یک روش استفاده از ثبات‌ها و روش دیگر استفاده از پشته است. درروش اول مقادیری که باید به‌عنوان پارامتر منتقل شوند، در برنامه پاسکال در ثبات قرار می‌گیرند و در زبان اسمبلی از این ثبات‌ها استفاده می‌شود.

مثال) برنامه‌ای که با استفاده از زیر برنامه پاسکال، مکان‌نما را به سطر و ستون خاصی از صفحه‌نمایش منتقل می‌کند. در این برنامه، سطر و ستون موردنظر، در برنامه پاسکال در ثبات قرار می‌گیرند و در برنامه اسمبلی از آن‌ها استفاده می‌شود.

در این برنامه از زیر برنامه cls که در مثال قبل مطرح شد برای پاک کردن صفحه‌نمایش استفاده می‌شود. سپس مکان‌نما به سطر و ستون خاصی منتقل‌شده، متنی در آنجا نوشته می‌شود. ضمناً سطر و ستون از ورودی خوانده می‌شوند (سطر = row و ستون = col).

برنامه پاسکال

```
Program test;
        Uses dos;
        Var     row, col: integer;
        Procedure locate; far; external; {$l locate.obj{
        Procedure cls; far; external; {$l cls.obj{
Begin {test}
        Cls;
        Write ('Enter row (0-24):');
        Readln (row);
        Write ('Enter col (0-79):');
        Readln (col);
        Asm
        Mov     bx, row
        Mov     dx, col
        Mov     dh, bl
        End;
        Locate;
        Write ('new location is here;('!
        Readln;
end.
```

برنامه LOCATE اسمبلی

```
Cseg segment para public 'code'
     Public locate
     Assumecs: cseg
Locate       proc    far
     Mov     ah, 2h
     Mov     bh, 0
     Int     10h
     Retf
Locate       endp
Cseg ends
     End
```

عملکرد برنامه

پس از اجرای برنامه، سطر و ستون موردنظر درخواست می‌شود که باید آن‌ها را وارد کنید. پس از واردکردن سطر و ستون، مکان‌نما به آن محل منتقل می‌شود و پیام "New location is here" تایپ می‌شود.

زمانی که تعداد پارامترها زیاد باشد برای انتقال پارامترها از پشته استفاده می‌کنیم، زیرا تعداد ثبات‌ها در کامپیوتر محدود است. در این روش پارامترها به‌صورت زیر در پشته ذخیره می‌گردند.

اگر زیر برنامه اسمبلی (فراخوانی شونده)، از نوع far باشد:

0 آدرس اشاره‌گر پشته فراخوان

02 آدرس آفست برنامه فراخوان

04 آدرس سگمنت برنامه فراخوان (برای برگشت به برنامه فراخوان)

06 آدرس آخرین پارامتر

08 یکی مانده به آخرین پارامتر

.

.

.

اگر نوع زیر برنامه near باشد، چون آدرس سگمنت کد، در پشته ذخیره نمی‌شود، پارامترها به‌صورت زیر در پشته ذخیره می‌گردند:

00 آدرس اشاره‌گر پشته برنامه فراخوان

02 آدرس آفست برنامه فراخوان

04 آخرین پارامتر

06 یکی مانده به آخرین پارامتر

.

.

.

چون زبان اسمبلی از ثبات BP استفاده می‌کند، مجبوریم که ثبات BP را در ابتدای زیر برنامه اسمبلی، جهت برگشت به برنامه پاسکال، در پشته ذخیره کنیم. برای این کار در ابتدای برنامه اسمبلی دستور Push BP را داریم که مقدار BP را در ابتدای پشته ذخیره می‌نماید. سپس محتویات SP (که عنصر مربوط به بالای پشته را آدرس‌دهی می‌نماید) را در ثبات BP ذخیره می‌نماییم. حال با اشاره‌گر BP می‌توان به پارامترهایی که توسط پاسکال به اسمبلی انتقال یافت دست‌یافت. به‌طوری‌که محتوی [BP+06] آخرین پارامتر، محتوی [BP+08] یکی مانده به آخرین پارامتر و ... ولی برای زیر برنامه‌های نوع near، محتوی [BP+04] آخرین پارامتر و محتوی [BP+06] یکی مانده به آخرین پارامتر و ... است.

پس پارامترها در زبان پاسکال از چپ به راست در پشته ذخیره می‌گردند و از راست به چپ از پشته قابل بازیابی می‌باشند. به‌عنوان‌مثال اگر برنامه پاسکال دو پارامتر را به برنامه اسمبلی انتقال دهد، پارامتر اول را در برنامه اسمبلی می‌توان از آدرس [BP+08] و پارامتر دوم را می‌توان از آدرس [BP+06] بازیابی نمود (برای زیر برنامه اسمبلی نوع far). ولی اگر نوع زیر برنامه اسمبلی near باشد در این صورت محتوی [BP+06] پارامتر دوم و محتوی [BP+04] پارامتر اول است.

مثال) برنامه‌ای که با استفاده از پارامترهایی که در پشته قرار می‌گیرند، برنامه اسمبلی را فراخوانی می‌کند و مکان‌نما را به سطر و ستون خاصی منتقل می‌نماید. در این برنامه از زیر برنامه Locatexy استفاده‌شده است.

همان‌طور که در برنامه ملاحظه می‌شود اولین پارامتر در آدرس [BP+08] قرار دارد که شماره سطر است و در ثبات DH قرار می‌گیرد. دومین پارامتر در آدرس [BP+06] قرار دارد که شماره ستون است و در ثبات DL قرار می‌گیرد تابع 02h وقفه 10h فراخوانی می‌شود تا عمل موردنظر صورت گیرد.

نکته:

در این برنامه، دو بایت از پشته توسط ثبات BP، دو بایت توسط سگمنت کد به دلیل استفاده از far و دو بایت توسط ip اشغال می‌شود. یعنی بایت‌های شماره ۰و۱و۲و۳و۴و۵ اشغال می‌شوند و از بایت شماره ۶ پارامترها قرار می‌گیرند.

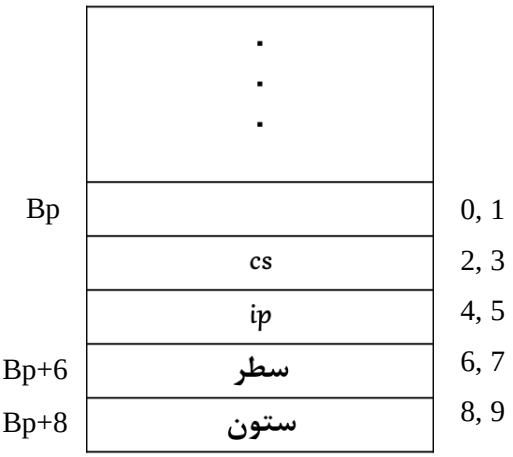
برنامه پاسکال

```
Program test;
      Uses dos;
      Var row, col: integer;
      Procedure locatexy (row, col: integer); far; external; {$l locatexy.obj}
      Procedure cls; far; external; {$l cls.obj}
Begin {test}
      Cls;
      Write ('Enter row (0-24) :');
      Readln (row);
      Write ('Enter col (0-79) :');
      Readln (col);
      Locatexy (row, col);
      Write ('new location is here!');
      Readln;
end.
```

برنامه Locatexy اسمبلی

```
Cseg segment  para  public  'code'
      Public  locatexy
      Assumecs: cseg
Locatexy proc         far
      Push    bp
      Mov     bp, sp
      Mov     bx, [bp + 8]
      Mov     dh, bl
      Mov     bx, [bp + 6]
      Mov     dl, bl
      Mov     ah, 2h
      Mov     bh, 0
      Int     10h
      Pop     bp
      Retf
Locatexy endp
Cseg ends
      End
```

فایل locatexy.obj باید در مسیر برنامه باشد.



۱۰-۳ ارتباط اسمبلی با زبان C

۱۰-۳-۱ دستورات اسمبلی در زبان C

برای نوشتن دستورات اسمبلی در زبان C از دستور asm به‌صورت زیر استفاده می‌شود:

```
Asm {
        دستورات اسمبلی
}
```

اگرچند دستور اسمبلی در یک سطر باشند، هر دستور باید به ; ختم شود وگرنه نیاز به ; نیست.

۱۰-۳-۲ استفاده از زیر برنامه‌های اسمبلی در برنامه C

برنامه‌نویسان ترجیح می‌دهند روال‌های اسمبلی را به‌صورت مجزا نوشته، به obj تبدیل کرده آن‌ها را در زبان C فراخوانی کنند. برای برقراری ارتباط بین C و اسمبلی باید سه نکته را رعایت نمود:

۱- نوع و ترتیب استفاده از سگمنت ها در توربو اسمبلر باید با نوع و ترتیب سگمنت ها در توربو C تطبیق کامل داشته باشد.

۲- برنامه‌های TC و TASM شناسه‌ها و توابع خود را به اشتراک‌گذارند.

۳- استفاده از TLINK برای برقراری ارتباط

۱۰-۳-۳ راهنمای model.

در توربو C شش مدل حافظه tiny، small، comact، medium، large و huge وجود دارد و در اسمبلی می‌توان با دستور راهنمای model. ، مدل موردنظر را انتخاب کرد.

۱۰-۳-۴ کوچک و بزرگ بودن حروف و متغیرها

در برخی از نسخه‌های C، بین حروف بزرگ و کوچک تفاوت است ولی توربو اسمبلر بین حروف بزرگ و کوچک فرقی قائل نمی‌شود. لذا در ارتباط بین اسمبلی و C به این موضوع توجه داشته باشید.

۱۰-۳-۵ پیش‌فرض سگمنت

در برخی از زیر برنامه‌های اسمبلی، نیاز داریم انواع سگمنت داده را تعریف نماییم. هنگام ورود از برنامه C به برنامه اسمبلی، توربو C ثبات‌های CS و DS را با پیش‌فرض معینی مقداردهی می‌کند.

در تمام مدل‌های حافظه ting، small و medium ثبات‌های SS و DS مقادیر یکسانی را به خود می‌گیرند.

MODEL	CS	DS
Ting	_text	group
Small	_text	group
Compact	_text	group
Medium	Filename_text	group
Large	Filename_text	group
Huge	Filename_text	calling_filename_date

۱۰-۳-۶ ارتباط شناسه‌های external و public در توربو C و توربو اسمبلر

توربو اسمبلر می‌تواند به شناسه‌های عمومی و توابع خارجی توربو C دسترسی پیدا کند و برعکس. در این صورت TC انتظار دارد تمام شناسه‌ها و توابع عمومی در اسمبلی با خط (_) شروع شود و ما باید تمام شناسه‌ها و توابع خارجی C را در اسمبلی با خط ربط شروع نماییم. (با استفاده از راهنمای U– در توربو اسمبلر (TASM) می‌توان خط ربط را از ابتدای شناسه‌ها حذف کرد). اگر شناسه‌ای برای یک برنامه خارجی باشد رد اسمبلی باید به‌صورت extrn تعریف شود و اگر برنامه اسمبلی در TC فراخوانی می‌گردد این برنامه در TC باید به‌صورت extrn تعریف شود و در اسمبلی باید به‌صورت public تعریف گردد.

مثال) برنامه‌ای که در این مثال، تغیر i در برنامه C تعریف می‌شود. برنامه اسمبلی به آن‌یک واحد اضافه می‌کند و برنامه C مقدار جدید i را چاپ می‌کند.

توضیحات:

زیر برنامه INCI در C به‌صورت extrn تعریف‌شده و در برنامه اسمبلی در جلوی آن خط ربط قرار گرفت و از نوع far تعریف گردید. متغیر i در C به‌صورت متغیر عمومی (خارج از ()main) و در اسمبلی به‌صورت extrn، با خط ربط و از نوع word تعریف شد (چون نوع متغیر در C، int تعریف‌شده است). در برنامه مقدار i برابر با صفر است که در برنامه اسمبلی یک واحد به آن اضافه‌شده است، درنتیجه، خروجی برنامه C به‌صورت i=1 است.

برای اجرای برنامه کافی است در خط فرمان دستور زیر تایپ شود:

TCC -MC C نام برنامه اسمبلی.ASM نام برنامه

پس از اجرای این دستور، یک فایل EXE همنام با فایل برنامه C ایجاد می‌شود که با تایپ نام برنامه اجرایی (نام برنامه C)، برنامه اجرا می‌شود.

برنامه C

```
#include <stdio.h>
extern      void    far     inci();
extern      void    far     cls(void);
int   i=0;
void  main(void){
      inci();
      printf("\n       i=%i",i);
}
```

برنامه اسمبلی

```
.model      small
.data
extrn _i:word
.code
Public      _inci
_inci proc
      Inc _i
      Retf
      Endp
_inic endp
end
```

۱۰-۳-۷ ترجمه چند فایل C و اسمبلی

برای ترجمه چند فایل C و اسمبلی، دو روش وجود دارد. درروش اول از فرمان TCC به‌صورت زیر استفاده می‌شود:

TCC-MS C1 نام فایل C2 نام فایل ... Cn نام فایل ۱ اسمبلی نام فایل ۲ اسمبلی نام فایل ... اسمبلی نام فایل n

در این روش ابتدا نام فایل‌های C را بدون پسوند در جلوی TCC قرار می‌دهیم و سپس نام فایل اسمبلی را با پسوند ASM، بعدازآن‌ها ذکر می‌نماییم. بدین ترتیب یک فایل EXE به نام اولین فایل C ایجاد می‌گردد.

درروش دوم، فایل‌ها را در TC در فایل پروژه قرار می‌دهیم و فایل‌های اسمبلی باید قبلاً توسط TASM یا MASM به فایل‌های obj ترجمه گردند.

۱۰-۳-۸ انتقال پارامترها بین اسمبلی و TC

در اینجا دو موضوع به شرح ذیل موردبحث قرار می‌گیرند:

۱- انتقال اطلاعات از برنامه C به اسمبلی

۲- بازگرداندن مقادیر بازگشتی از برنامه اسمبلی به برنامه C

۱۰-۳-۹ ارسال پارامترها از برنامه C به اسمبلی

برای ارسال پارامترها از برنامه C به برنامه اسمبلی، دو روش وجود دارد:

روش اول استفاده از ثبات‌ها

روش دوم استفاده از پشته

مثال) برنامه‌ای که ابتدا یک نام را از ورودی می‌خواند و یک ناحیه از صفحه‌نمایش را پاک می‌کند و نام خوانده‌شده را وسط این ناحیه چاپ می‌کند. هدف از این برنامه آشنایی با انتقال پارامترها باثبات و دستور ASM است.

```
#include <stdio.h>
#include <conio.h>
extrn void far cls(void);
void main(void){
      char     fname[21];
      printf ("\nplease enter name : ");
      gets(fname);
      clrscr();
      asm{
            mov     al, 10
            mov     ch, 5
            mov     cl,10
            mov     dh, 15
            mov     dl, 40
            mov     bh, 17h
      }
      Cls();
      asm{
            mov     ah, 02
            mov     dh, 10
            mov     dl, 25
            mov     bh, 0
            int     10h
      }
      printf("%s",fname);
      getch();
      clrscr();
}
```

برنامه اسمبلی

```
IGROUP    group   _TEXT                ;       program         segment
DGROUP    group   _BSS, _DATA          ;       data segment
      Assumecs: IGROUP, ds: DGROUP, es: DGRPUP, ss: DGROUP
_BSS      segment       word    public  'BSS'
_BSS      ends
_DATA     segment       word    public  'DATA'
_DATA     ends
_TEXT     segment       byte    public  'CODE'
      Public  _cls
_cls  proc    far
      Mov     ah, 06
      Int     10h
      Ret
_cls  endp
_TEXT     ends
End
```

۱۰-۳-۱۰ استفاده از پشته برای انتقال پارامترها

در زبان C برخلاف بقیه زبان‌ها پارامترها از سمت راست به چپ‌روی پشته ذخیره می‌گردند (به‌طوری‌که ابتدا سمت راست‌ترین پارامتر روی پشته ذخیره می‌گردد). دستور زیر را در نظر بگیرید:

lovatexy (row,col);

این دستور، کدهای زیر را ایجاد می‌کند:

```
push word ptr       dgroup :_col

push word ptr       dgroup :_row

call        locatexy

ADD         SP,4
```

در ابتدای برنامه اسمبلی باید BP را در پشته ذخیره کرد تا بتوان از برنامه اسمبلی به برنامه C برگشت کرد. سپس در اسمبلی SP را در BP قرار می‌دهیم. در این صورت برای دسترسی به پارامترها در برنامه اسمبلی، دو حالت پیش می‌آید. در حالتی که برنامه اسمبلی near است، پارامترها به‌صورت زیر در پشته ذخیره می‌گردند:

00 آدرس اشاره‌گر پشته فراخوان

02 آدرس آفست برگشت به برنامه فراخوان

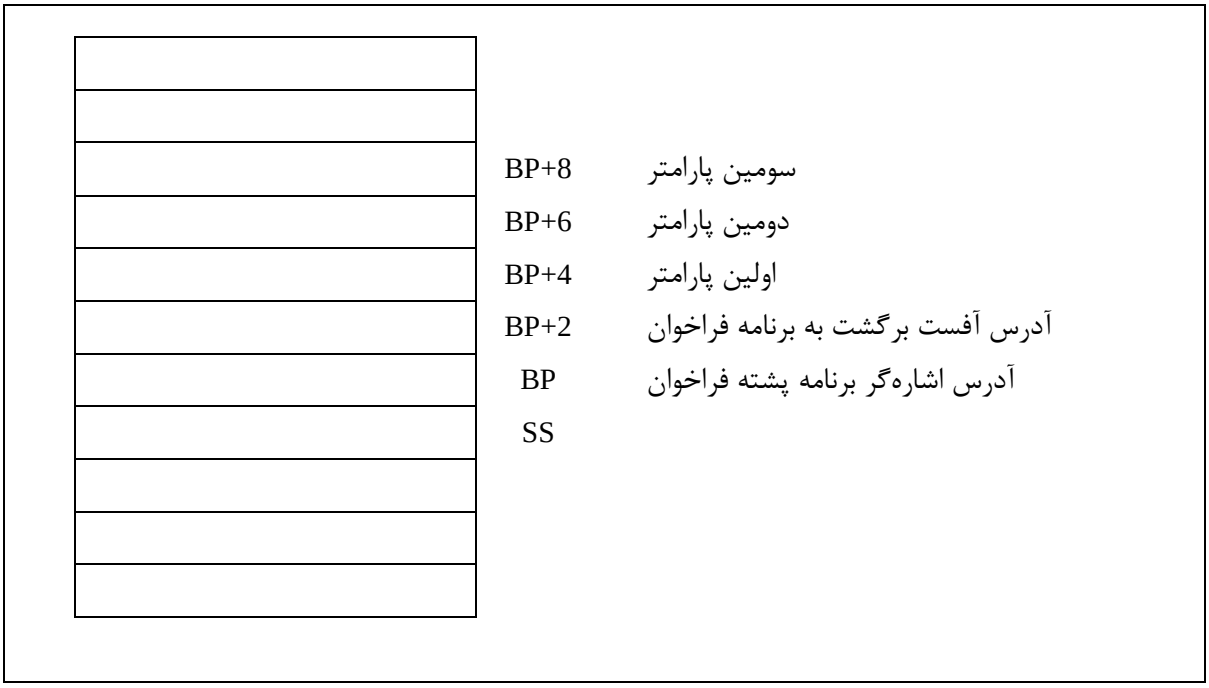
04 اولین پارامتر

06 دومین پارامتر

.

.

.



مثال) برنامه‌ای که یک کاراکتر در برنامه C و تعداد دفعات چاپ را از ورودی می‌خواند و با استفاده از یک برنامه اسمبلی، کاراکتر خوانده‌شده را به تعداد معین نمایش می‌دهد (برنامه اسمبلی از نوع near است).

توضیح

برنامه اسمبلی disp در C به‌صورت extrn تعریف شد و در برنامه اسمبلی به‌صورت public و از نوع near تعریف شد. برنامه C با یک پیام، کاراکتر را می‌خواند و در ch قرار می‌دهد و بعدازآن، تعداد دفعاتی که کاراکتر باید چاپ شود خوانده‌شده در شناسه count قرار می‌گیرد. سپس برنامه اسمبلی disp با دو پارامتر ch و count فراخوانی می‌شود. چون در برنامه اسمبلی، disp از نوع near است، برای دسترسی به پارامتر اول محتویات [BP+4] به ثبات al (کاراکتری که باید نمایش داده شود) و محتویات پارامتر دوم، [BP+06]، به ثبات CX منتقل‌شده است (تعداد دفعاتی که کاراکتر باید نمایش داده شود).

با استفاده از تابع 0eh وقفه 10h موجود در ثبات al صفحه‌نمایش چاپ‌شده است. چون دستور INT 10H به تعداد ۱۰ بار فراخوانی شد، کاراکتر موجود در ثبات AL به تعداد CX بار، در صفحه‌نمایش، چاپ می‌شود.

برنامه C

```
#include <stdio.h>
#include <conio.h>
Extern void disp(char, int);
Void main(void)
{
        char ch;
        int        count;
        printf("\nPlease ente a char:");
        scanf("%c",&ch);
        printf("\nPlease count for display char:");
        scanf("%i",&count);
        disp(ch, count);
}
```

برنامه اسمبلی

```
.model small
.data
.code
Public          _disp
_dispproc       near
        Push    bp
        Mov     bp, sp
        Mov     al, [bp+04]
        Mov     cx, [bp+06]
        Mov     ah, 0eh
Loop1:          Int     10h
        Loop    loop1
        Pop     bp
        Ret     4
_dispendp
End
```

خروجی:

```
Please enter a char:    B
Please enter count for display char: 12
BBBBBBBBBBBB
```

در حالت دوم، چون نوع برنامه اسمبلی far است، پارامترها به‌صورت زیر در پشته قرار می‌گیرند:

00 آدرس اشاره‌گر پشته فراخوان

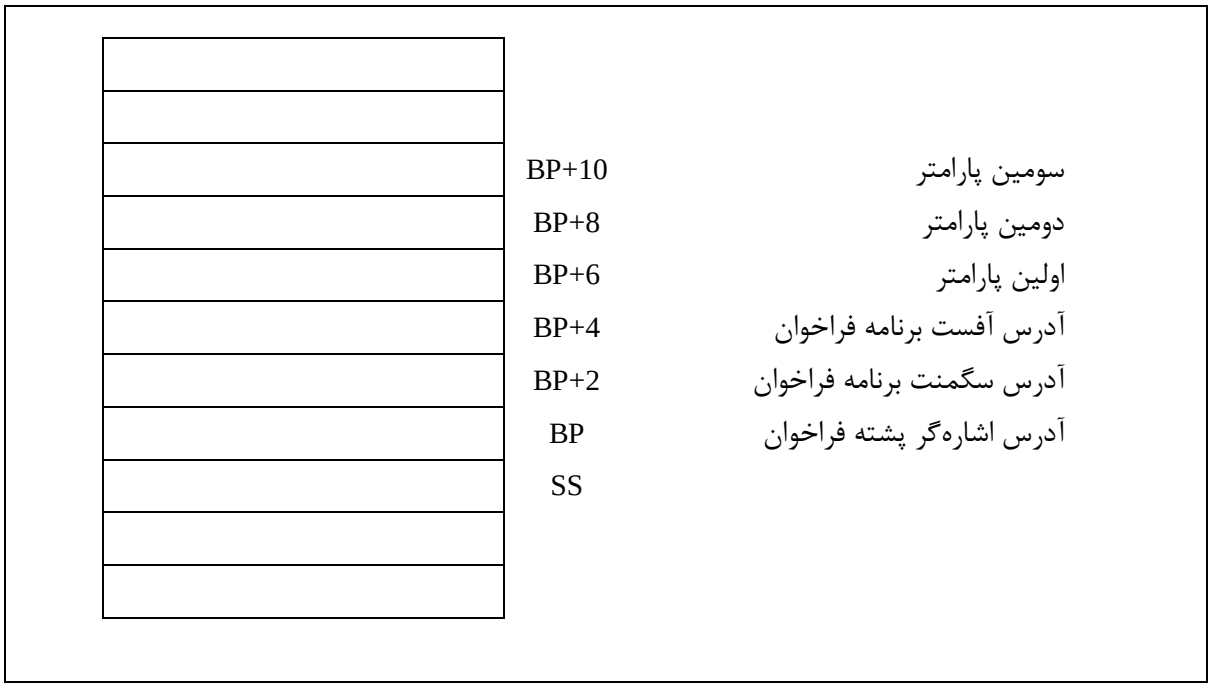
02 سگمنت کد برنامه فراخوان

04 آفست برنامه فراخوان

06 پارامتر اول

08 پارامتر دوم

پارامتر سوم



مثال) برنامه‌ای که سطر و ستون را از ورودی خوانده با استفاده از یک برنامه اسمبلی به نام locatexy مکان‌نما را به سطر و ستون خواسته‌شده انتقال می‌دهد و یک خروجی را در آن سطر و ستون چاپ می‌کند. زیر برنامه locatexy از نوع far است.

```
#include <stdio.h>
#include <conio.h>
extern void far locatexy (int, int);
void main (void) {
        Int       row, col;
        clrscr();
        printf("\nPlease enter row (1-24):");
        scanf("%i",&row);
        printf("\nPlease enter col (1-79):");
        scanf("%i",&col);
        clrscr();
        locatexy(row,col);
        printf("link a moudle c with assambly (far)");
}
```

برنامه اسمبلی Locatexy

```
IGROUP      group   _TEXT           ;program        segment
DGROUP      group   _BSS,_Data      ;data segement
        Assumecs:IGROUP, ds:DGROUP, es:DGROUP, ss: DGROUP
        _BSS    segment         segment         word    public
        _BSS    ends
        _DATA           segment         word    public  'DATA'
        _DATA           ends
        _TEXT segment           byte    public  'CODE'
        Public  _locatexy
_locatexy       proc    far
        Push    bp
        Mov     bp, sp
        Mov     dh, [bp+06]
        Mov     dl, [bp+08]
        Mov     ah, 02
        Xor     bh, bh
        Int 10h
        POP     BP
        Retf    4
_locatexy       endp
_TEXT           ends
        End
```

همان‌طور که در برنامه اسمبلی ملاحظه می‌شود برای دسترسی به پارامتر اول، محتویات [BP+06] را در ثبات dh و برای دسترسی به پارامتر دوم، محتویات [BP+08] را در dl قرار می‌دهیم. با اجرای تابع شماره ۲ وقفه 10h مکان‌نما به سطری که مقدار آن در dh و ستونی که مقدار آن در ثبات dl قرار دارد، انتقال می‌یابد.

برای برگشت از برنامه اسمبلی به برنامه C، دستور Ret 4 در انتهای برنامه اسمبلی قرار دارد (4 به معنی این است که چهار بایت از پشته بازیابی می‌شود). زیرا دو پارامتر دو بایتی از نوع int، در هنگام فراخوانی برنامه

اسمبلی، در پشته قرار گرفتند. قبل از بازگشت از برنامه اسمبلی، این دو پارامتر باید از پشته بازیابی شوند تا اختلال در خاتمه برنامه پیش نیاید.

۱۰-۳-۱۱ بازگردان مقادیر از اسمبلی به C

برای بازگردان مقادیر یک بایتی و دو بایتی از ثبات AX و چهار بایتی از ثبات DX:AX استفاده می‌شود و مقادیر بیش از چهار بایت را به‌صورت static ذخیره می‌نماییم و آنگاه اشاره‌گری به آن‌ها بازگشت می‌دهیم. نوع اطلاعات و نحوه برگشت مقادیر در جدول زیر آورده شده است.

نوع مقدار بازگشتی	ثبات
Unsigned char	AX
Char	AX
Enum	AX
Unsigned short int	AX
Short int	AX
Int	AX
Unsigned int	AX
Long	DX:AX
Float, double, long double	8087 top of stack
Near *	AX
far *	DX:AX

فصل یازدهم پیوست ها

۱-۱۱ برنامه اشکال‌زدایی DEBUG

Debug وسیله‌ای جهت اشکال‌زدایی، اجرا و تغییر در برنامه‌های اسمبلی است و بیشتر برای کسانی مفید است که با زبان اسمبلی به‌طور تخصصی کار می‌کنند. این برنامه دارای خصیصه‌هایی است که به‌آسانی می‌توان از آن استفاده نمود. بعضی از اعمالی که برنامه debug می‌تواند انجام دهد عبارت‌اند از:

۱- تست برنامه‌ها
۲- انتقال فایل‌ها به حافظه، مشاهده محتویات آن‌ها و اعمال تغییرات در آن‌ها
۳- اجرای برنامه‌ها و فرمان‌های سیستم‌عامل DOS
۴- خواندن سکتورهایی از دیسک و یا نوشتن محتویات جدیدی در آن‌ها
۵- ایجاد و اجرای برنامه‌هایی به زبان اسمبلی

یکی از مزایایی که نوشتن برنامه‌ها به زبان اسمبلی در debug دارد این است که مستقیماً قابل‌اجرا بوده و نیازی به ترجمه توسط اسمبلر ندارند.

چون اعداد و ارقام و همچنین انجام محاسبات در debug در مبنای ۱۶ می‌باشند، لذا آشنایی مختصری با چگونگی انجام محاسبات در مبنای ۱۶ ضروری است.

زیر برنامه debug به‌صورت زیر استفاده می‌شود:

[پارامترها] [نام فایل] [مسیر۲] debug [مسیر۱]

مسیر۱، محل وجود برنامه debug را مشخص می‌کند که اگر منظور نشود، مسیر جاری و یا مسیرهای اعلام‌شده توسط فرمان path منظور خواهند شد.

مسیر۲، محل وجود فایل را مشخص می‌کند که برنامه debug باید روی آن کار کند.

پارامترها اسامی آرگومان‌هایی هستند که باید به فایل موردنظر (فایلی که debug روی آن کار می‌کند) منتقل شوند.

پس از اجرای برنامه debug اعمال ذیل انجام می‌شود.

۱- ثبات‌های ناحیه CS، DS، ES و SS به آدرس اولین ناحیه بعد از برنامه debug اشاره می‌کند.
۲- ثبات اشاره‌گر دستور (IP) به آدرس 100H (اولین دستور بعد از PSP) اشاره می‌کند.
۳- ثبات اشاره‌گر پشته به آخرین ناحیه یا قسمت موقت فایل command.com اشاره می‌کند.
۴- محتوی سایر ثبات‌های عمومی صفر خواهد شد و ثبات فلگ شامل محتویات ذیل خواهد بود:

NV UP EI PL NZ NA PO NC

۱-۱-۱۱ دستورات debug

برنامه debug دارای تعدادی فرمان مختص به خود است که در ذیل به بررسی آن‌ها می‌پردازیم. این دستورها غالباً یک کاراکتری هستند:

۱۱-۱-۱-۱ دستور R (Register)

این دستور برای نمایش محتویات ثبات‌ها مورداستفاده قرارگرفته به‌صورت زیر به کار می‌رود:

R [اسم ثبات]

اسم ثبات، مشخص می‌کند که محتویات چه ثباتی باید نمایش داده شود. اگر اسم ثبات ذکر نشود محتویات کلیه ثبات‌ها در صفحه‌نمایش ظاهر خواهد شد. لازم به ذکر است که محتویات ثبات‌ها در مبنای ۱۶ بیان خواهند شد. پس از نمایش محتویات ثبات‌ها، غلامت : (کولن) ظاهرشده کامپیوتر منتظر دریافت محتویات جدید ثبات‌ها خواهد بود. اگر کاربر کلید Enter را فشار دهد محتویات ثبات بدون تغییر باقی می‌ماند و اگر کاربر یکتا چهار رقم مبنای ۱۶ را وارد نماید و سپس کلید Enter را فشار دهد این مقدار به ثبات نسبت داده می‌شود. اسامی ثبات‌هایی که می‌توانند در این دستور ظاهر شوند عبارت‌اند از:

AX	BP	SS
BX	SI	CS
CX	DI	IP
DX	DS	PC
SP	ES	F

مثال)

نمونه‌های از کاربرد دستور R.

توضیح

این مثال مقدار ثبات M را نمایش می‌دهد و مقدار جدید را دریافت می‌کند.

```
A > DEBUG CHKDSK.COM
-r
Ax = 0000   BX = 0000   CX = 243B   DX = 0000   SP = FFFE   BP = 0000
SI = 0000   DI = 0000   DS = 40EB   ES = 40EB   SS = 40EB   CS = 40EB
IP = 0000            NV   UP   EI   NZ   NA   PO   NC
40EB: 0100  E9   C523   JMP   24CB
-r   IP
IP   0101
-
```

۱۱-۱-۱-۲ دستور H (Hexarithmetic)

چون در برنامه debug کلیه اعداد و ارقام در مبنای ۱۶ است و محاسبات نیز در مبنای ۱۶ انجام می‌شوند، کار کردن با آن، کمی دشوار است. دستور H که برای انجام محاسبات در مبنای ۱۶ به کار می‌رود می‌تواند در این زمینه به برنامه‌نویس کمک کند. این دستور به‌صورت زیر به کار می‌رود:

H <مقدار> <مقدار>

دو مقداری که در این دستورات ذکر می‌شوند در مبنای ۱۶ می‌باشند. دستور H حاصل جمع و تفریق این دو مقدار را در مبنای ۱۶ محاسبه کرده و در صفحه‌نمایش نشان می‌دهد. ابتدا حاصل جمع و سپس حاصل تفریق دو مقدار نشان داده می‌شود.

مثال) برنامه‌هایی از کاربرد دستور H

```
-H  12      10
0022 0002
-
H   FF      FF
01FE0000
-
-H  IA      4F
00F0 0052
-
```

۱۱-۱-۱-۳ دستور N (Name)

با این دستور دو کار را می‌توان انجام داد:

۱- انتخاب نام فایل جهت انتقال آن به حافظه و یا انتقال محتویات حافظه به آن.

۲- انتخاب پارامترهایی برای فایلی که debug روی آن کار می‌کند.

این دستور به‌صورت زیر به کار می‌رود:

1. N [مسیر] [فایل]

2. N [مسیر] [اسامی فایل‌ها]

مسیر، محل وجود فایل‌ها را مشخص می‌کند. نحوه کاربرد اول برای هدف اول و نحوه کاربرد دوم برای هدف دوم مورداستفاده قرار می‌گیرد.

مثال)

نمونه‌ای از کاربرد دستور N

```
-N  file1.exe
-L
-N  file1.dat       file2.dat
-G
```

در این مثال، فایل file1.exe انتخاب و توسط دستور L به حافظه منتقل می‌شود. دستور سوم فایل‌های file1.dat و file2.dat را به‌عنوان پارامترهای فایل file1.exe انتخاب می‌کند.

اگر پس از دستور G، دستور W را صادر کنیم، این دستور موجب می‌شود تا محتویات فایل file1.exe در فایل file2.dat نوشته شود. برای جلوگیری از این حالت، توصیه می‌شود که قبل از اجرای هر دستور L و یا W نام فایل مناسب را توسط دستور N انتخاب کنید.

۱۱-۱-۱-۴ دستور Q (Quit)

این دستور برای خروج از برنامه debug، بدون ثبت تغییرات اعمال‌شده به فایل موردنظر، به کار می‌رود و به‌صورت زیر استفاده می‌شود:

Q

۱۱-۱-۱-۵ دستور G (GO)

این دستور موجب اجرای برنامه‌ای که اکنون توسط debug به حافظه منتقل‌شده است می‌شود و به‌صورت زیر به کار می‌رود:

G [= آدرس] [آدرس]

اگر دستور G بدون آدرس‌های لازم به‌کاربرده شود فایل موجود در حافظه، همان‌طور که در سطح سیستم‌عامل اجرا می‌گردد، در debug نیز اجرا خواهد شد. اولین آدرس که با علامت "=" شروع می‌شود، به debug می‌گوید که برنامه را از چه آدرسی اجرای نماید. اگر آدرس‌ها همراه با علامت مساوی (=) نباشد، به‌عنوان آدرس‌های توقف محسوب می‌شوند. حداکثر ۱۰ نقطه توقف قابل‌تعریف است. پس‌ازاینکه اجرای برنامه به نقطه توقف رسید، اجرای آن متوقف‌شده محتویات ثبات‌ها مشابه حالتی که از فرمان R استفاده‌شده باشد روی صفحه‌نمایش ظاهر خواهد شد. وقتی اجرای برنامه به یک نقطه توقف بعدی نادیده گرفته خواهند شد. اگر پس از رسیدن به نقطه توقف، از دستور G استفاده کنیم، اجرای برنامه از این نقطه تا انتهای آن ادامه پیدا می‌کند.

مثال) نمونه‌ای از کاربرد دستور G

```
_GCS : 7550
```

دستور فوق موجب اجرای برنامه موجود در حافظه تا آدرس 7550 از ناحیه CS می‌شود و سپس محتویات جدید ثبات‌ها را نمایش می‌دهد.

۱۱-۱-۱-۶ دستور I (Input)

این دستور برای خواندن یک بایت اطلاعات از پورتی که شماره آن در این دستور ذکر می‌شود به‌کاررفته و به‌صورت زیر استفاده می‌شود:

I <شماره پورت>

شماره پورت، مشخص‌کننده پورتی است که اطلاعات باید ازآنجا خوانده شوند و آدرس پورت حداکثر یک عدد ۱۶ بیتی است.

مثال) با اجرای دستور زیر، یک بایت از پورت شماره 2F8 خوانده‌شده در صفحه‌نمایش ظاهر می‌گردد.

```
I 2F8
```

۱۱-۱-۱-۷ دستور O (Output)

این دستور برای فرستادن یک بایت از اطلاعات به یک پورت به‌کاررفته و به‌صورت زیر استفاده می‌شود:

O <شماره پورت> <مقدار>

شماره پورت، که حداکثر یک عدد ۱۶ بیتی است مشخص‌کننده پورتی است که اطلاعات (<مقدار>) موردنظر که حداکثر یک بایت است باید به آن منتقل شود.

مثال) دستور زیر مقدار 4E را به پورت شماره 2F8 منتقل می‌کند.

```
O 2F8 4F
```

۱۱-۱-۱-۸ دستور D (Dump)

این دستور برای نمایش محتویات قسمت‌هایی از حافظه بر روی صفحه‌نمایش به کار می‌رود و به‌صورت زیر استفاده می‌شود:

```
D [محدوده حافظه]
```

محدوده حافظه ممکن است به‌صورت های زیر باشد:

```
آدرس۱[,]آدرس۲
آدرس L <مقدار>
```

آدرس۱ و آدرس۲ محدوده‌ای را مشخص می‌کنند که دستور D باید روی آن‌ها عمل کند. آدرس اول می‌تواند یک ناحیه باشد مثل CS:100

آدرس محلی از حافظه را مشخص می‌کند که اجرای فرمان D باید ازآنجا شروع شود و L همراه با <مقدار> مشخص‌کننده تعداد بایت است که فرمان D باید محتویات آن‌ها را نمایش دهد.

اگر D را بدون "محدوده حافظه" به کار ببریم، ۱۲۸ بایت از ابتدای برنامه و یا اولین آدرسی که در آخرین دستور D منظور شده است نمایش داده می‌شود.

مثال)

```
A > debug
_F000 : FFF5, FFFC
F000 : FFF0  30    31    2F_31  30    2F    38    34    01/10/84
_D   F000: FFF5,L,8
F000 : FFF5  30    31    2F_31  30    2F    38    34    01/10/84
```

۱۱-۱-۱-۹ دستور F (FULL)

این دستور برای پر کردن محدوده‌ای از حافظه با مقادیر خاصی به‌کاررفته و به‌صورت زیر استفاده می‌شود:

```
F_ <محدوده حافظه> <مقادیر>
```

"محدوده‌ای از حافظه" که توسط دو آدرس و یا یک آدرس و یک مقدار، که تعیین‌کننده تعداد بایت‌های حافظه است (مشابه آنچه که در دستور D گفته شد) مشخص می‌شود، تعیین می‌کند که چه قسمت‌هایی از حافظه باید توسط این دستور، محتویات جدید بگیرند.

<مقادیر>، شامل یک یا چند بایت و یا یک عبارت رشته‌ای است که "محدوده حافظه" مشخص‌شده، باید با این بایت‌ها و یا عبارت رشته‌ای پر شود. عبارت رشته‌ای باید در داخل نقل‌قول (‘’) قرار داشته باشد.

صفحه‌نمایش رنگی IBM محدوده‌ای از حافظه است که از آدرس B800H شروع می‌شود. اجرای دستوری که در مثال زیر آمده است موجب پر کردن صفحه‌نمایش با حرف K می‌شود:

مثال)

```
_F   B800   : 0 L 4000 “K”, 7
```

اگر تعداد بایت‌های محدوده آدرس، از تعداد مقادیری که ذکرشده کمتر باشد بقیه مقادیر منظور نخواهند شد. اگر تعداد بایت‌های محدوده آدرس از تعداد مقادیری که ذکرشده بیشتر باشد، بقیه محدوده حافظه با تکرار بایت‌های ذکرشده (از اولین بایت به آخرین بایت مقادیر ذکرشده) پر خواهند شد.

مثال)

```
_F   04BA  : 100   L100   42     52     54     41
```

با اجرای این دستور، محدوده حافظه 04BA : 100 تا 04BA : 1FF با بایت‌های مشخص‌شده پر می‌شود و سپس این مقادیر برای پر کردن بقیه 100H بایت تکرار می‌شوند.

مثال) دستور زیر 20H (32)بایت از حافظه را با شروع از آدرس 200H، با رشته Help پر می‌کند.

```
_F   200    L20    “Help”
```

مثال) دستور زیر از آدرس 210H تا آدرس 230H را با رشته listing پر می‌کند.

```
_F   210    L      230    “listing”
```

۱۱-۱-۱-۱۰ دستور L (Load)

این دستور برای انتقال سکتورهایی از دیسک به حافظه به کار می‌رود و به‌صورت زیر استفاده می‌شود:

L [آدرس] [درایور ۱ سکتور ۲ سکتور]

آدرس، محلی از حافظه را مشخص می‌کند که سکتورهای خوانده‌شده از دیسک باید ازآنجا به بعد ذخیره شوند. درایو، شماره درایوی است که مشخص می‌کند اطلاعات باید از آن درایو خوانده شوند. شماره صفر مشخص‌کننده درایو A و شماره یک مشخص‌کننده درایو B، شماره ۲ مشخص‌کننده درایو C و ... است.

سکتور ۱، مشخص‌کننده اولین سکتوری است که خواندن اطلاعات از روی دیسک باید ازآنجا شروع شود. سکتور۲، مشخص‌کننده تعداد سکتورهایی است که با شروع از سکتور۱ باید از روی دیسک خوانده شود.

مثال)

```
_L   100    0      0      1
```

اجرای دستور فوق موجب انتقال سکتور صفر از درایور A به محل 100H حافظه می‌شود. (100 مشخص‌کننده آدرس، اولین صفر مشخص‌کننده درایوA، دومین صفر مشخص‌کننده سکتور شروع و یک مشخص‌کننده تعداد سکتورهایی است که باید خوانده شوند). این سکتور، همان سکتور را انداز (boot record) است.

مثال)

```
_D   103    L8
165A: 0100  -4D    53     44     4F     53_34  2E     30     MSDOS4.0
```

دستور L علاوه بر انتقال سکتورهایی از دیسک به حافظه، قادر است یک فایل را نیز به حافظه منتقل کند.

مثال) دستوری که فایل command.com را به حافظه منتقل می‌کند.

```
A >  debug
_N   command.com
_L
```

در دستور L به نکات زیر توجه کنید:

۱- اگر دستور L را بدون هیچ پارامتری به کار ببریم، فایلی که آدرس آن در آدرس CS:100 قرار دارد به حافظه منتقل خواهد شد.

۲- پس از اجرای دستور L ثبات‌های BX:CX حاوی تعداد بایت‌هایی هستند که به حافظه منتقل‌شده‌اند.

۱۱-۱-۱-۱۱ دستور W (Write)

این دستور برای انتقال محتویات حافظه بر روی دیسک به‌کاررفته و به‌صورت زیر استفاده می‌شود:

```
W [آدرس] [[سکتور ۲ سکتور۱ درایو]]
```

آدرس، مشخص‌کننده محلی از حافظه است که اطلاعات موجود در حافظه با شروع از آن آدرس باید به دیسک منتقل شوند.

درایو، شماره درایوی را مشخص می‌کند که محتویات حافظه باید در آن درایو نوشته شوند شماره صفر مشخص‌کننده درایوA، شماره یک مشخص‌کننده درایو B، شماره ۲ مشخص‌کننده درایو C و غیره است.

سکتور۱، محل شروع قرار گرفتن اطلاعات بر روی دیسک را مشخص می‌کند.

سکتور۲، تعداد سکتورهای روی دیسک با شروع از سکتور ۱ را مشخص می‌کند که اطلاعات موجود در حافظه باید در این سکتورها نوشته شوند.

اگرW را بدون هیچ پارامتری به کار ببریم محتویات حافظه از آدرس 100H به دیسک منتقل می‌شوند و تعداد بایت‌هایی که باید منتقل شوند باید در ثبات‌های BX:CX قرار گیرد.نکته‌ای که باید به آن توجه داشته باشیم این است که اگر قبل از اجرای دستور W از دستورات G (Go) و یا T (Trace) استفاده کرده باشیم باید ثبات‌های BX:CX را مجدداً مقداردهی کنیم.

مثال)

```
-W CS : 100 1 37 2B
```

اجرای این دستور موجب می‌شود تا محتویات حافظه از آدرس ۱۰۰ با شروع از سکتور شماره ۳۷ و به تعداد B۲ سکتور بر روی درایو B نوشته شود.

دستور write برای نوشتن محتویات یک فایل موجود در حافظه بر روی دیسک نیز به کار می‌رود.این فایل باید قبلاً توسط برنامه debug و یا دستور Lو N به حافظه منتقل‌شده باشد.

```
A > debug
_command.com
_L
_W
```

اجرای دستور L موجب انتقال فایل command.com به حافظه و دستورWموجب انتقال آن فایل از حافظه به دیسک می‌شود.

۱۱-۱-۱-۱۲ دستور S (Search)

این دستور برای جستجو در محدوده‌ای از حافظه به‌کاررفته و به‌صورت زیر استفاده می‌شود:

S< مقادیر> <محدوده حافظه>

محدوده حافظه ،مشخص‌کننده محل‌هایی از حافظه است که باید عمل جستجو در آن محل‌ها انجام شود.این محدوده ممکن است با دو آدرس و یا یک آدرس و یک مقدار که مشخص‌کننده تعداد بایت‌هایی از حافظه است ،مشخص شود.مقادیر، شامل یک یا چند بایت و یا عبارت رشته‌ای است که باید در محدوده مشخص‌شده مورد جستجو قرار گیرد.عبارت رشته‌ای باید در داخل نقل‌قول (") قرار گیرد.

مثال)

```
_S   100 ,L,100  “IBM”
4005: 0103
```

دستور فوق ۱۰۰ بایت از حافظه را با شروع از محل ۱۰۰ حافظه ،رشته IBM را مورد جستجو قرار می‌دهد و آدرس‌هایی که حاوی این رشته باشند نشان می‌دهد.

مثال)

```
_S   100   200   “IBM”
```

اجرای دستور فوق از محل ۱۰۰ تا ۲۰۰ حافظه را برای پیدا کردن رشته "IBM" مورد جستجو قرار می‌دهد.

۱۱-۱-۱-۱۳ دستور M (Move)

این دستور برای انتقال محتویات محدوده‌ای از حافظه به محل دیگری از حافظه به کار می‌رود و به‌صورت زیر استفاده می‌شود :

M<آدرس> <محدوده حافظه >

محدوده حافظه ،مشخص‌کننده محل‌هایی از حافظه است که محتویات آن باید به‌جای دیگری منتقل شوند. این محدوده ممکن است با دو آدرس یا یک آدرس و یک مقدار که مشخص‌کننده تعداد بایت‌هایی از حافظه است مشخص شود.آدرس،محلی از حافظه را مشخص می‌کند که عمل انتقال باید از آن محل به بعد انجام شود.

مثال)

با اجرای دستورات زیر ،صفحه‌نمایش با حرف K پر خواهد شد.

در این مثال، ابتدا محدوده‌ای از حافظه را با شروع از آدرس ۱۰۰ و به تعداد ۴۰۰۰ بایت را با حرف K پر می‌کنیم و سپس محتویات این محدوده از حافظه را به محلی از حافظه با شروع از آدرس B800 منتقل می‌کنیم. توجه داریم که این آدرس ،شروع محلی از حافظه است که برای صفحه‌نمایش رنگی مورداستفاده قرار می‌گیرد(اگر صفحه‌نمایشی که با آن کار می‌کنید تک‌رنگ باشد می‌توانید آدرس آن را B000 منظور کنید).

```
_F   100,L,  4000 K
_M   100,L,4000  B800
```

۱۱-۱-۱-۱۴ دستور E (Enter)

این دستور برای تغییر محتویات محل‌هایی از حافظه مورداستفاده قرار می‌گیرد و به‌صورت زیر استفاده می‌شود:

```
E    <آدرس> [<مقادیر>]
```

آدرس، محلی از حافظه را مشخص می‌کند که محتویات حافظه از آن محل به بعد باید عوض شوند. مقادیر، مشخص‌کننده بایت یا بایت‌هایی است که باید به‌عنوان محتویات جدید محل‌های حافظه منظور شود. این مقدار می‌تواند به‌صورت یک عبارت رشته‌ای نیز باشد. عبارت رشته‌ای باید در داخل نقل‌قول(") قرار گیرد. اگر مقادیر ذکر نشوند، محل‌های حافظه همراه با محتویات آن‌ها با شروع از آدرسی که مشخص می‌شود، در صفحه‌نمایش ظاهرشده محتویات جدید درخواست می‌شود.

مثال)

```
_L  100   0   0   1
_E  103   "Software"
_W  100   0   0   1
_
```

اجرای دستور فوق موجب می‌شود تا رکورد راه‌انداز (boot record) از درایو A به حافظه منتقل‌شده، مشخصه سیستم به "Software" عوض شود و مجدداً بر روی دیسک (در محل رکورد راه‌انداز) نوشته شود.

مثال)

```
_E  100
_40D5 : 0100      EB
```

اجرای دستور فوق محتویات محل 100H حافظه را که EB است نمایش داده و منتظر دریافت محتویات جدید این محل می‌شود:

```
_E  100
_40D5 : 0100      EB.25
```

با وارد نمودن عدد ۲۵ و فشار دادن کلید Enter، محتویات جدید محل 100H حافظه برابر ۲۵ خواهد بود. برای تغییر در محتویات محل‌های بعدی، به‌جای کلید Enter باید کلید Space فشار داده شود.

۱۱-۱-۱-۱۵ دستور A (Assemle)

با استفاده از این دستور می‌توانیم دستورات اسمبلی را ترجمه کنیم تا جهت اجرای آن دستورات، نیازی با اسمبلر نباشد. با دستور A می‌توانیم به‌راحتی برنامه‌های اسمبلی خود را در debug بنویسیم و سپس در فایل COM یا EXE ذخیره کنیم. این دستور به‌صورت زیر به کار می‌رود:

```
A    [آدرس]
```

آدرس محل شروع کار دستور A را مشخص می‌کند. اگر آدرس وارد نشود، آدرس شروع CS:100H منظور می‌شود، زیرا طولpsp ، برابر با 256 (100H) است. به همین دلیل debug مقدار 100H را در ثبات IP قرار می‌دهد و آخرین enter به debug می‌گوید که برنامه خاتمه یافت.

مثال) مجموعه دستورات زیر محتویات آدرس XXXX:010D را با مقدار 1AH جمع می‌کند و در آدرس 010DH قرار می‌دهد.

```
A     100
XXXX        : 0100  MOV   BX, [10D]
XXXX        : 104   ADD   BX, 1A
XXXX        : 107   MOV   [10D], BX
XXXX        : 10B   JMP   100
XXXX        : 10D   DW    3000
```

مثال)

```
A > debug test.com
_A 100
-40D5 : 0100 INT     5
-40D5 : 0102 MOV     AH, 0
-40D5 : 0104 INT     21
-40D5 : 0106
```

در مثال قبل محتویات فایل test.com با سه دستور اسمبلی نوشته‌شده است. اکنون می‌خواهیم این فایل را روی دیسک ذخیره کنیم:

```
_h    106     100
0206 0006
_R    CX
CX    0000
: 6
_W
Written       0006    bytes
```

در دستور اول، تعداد بایت‌های مصرفی توسط ۳ دستور فوق را مشخص کردیم. در دستور دوم محتویات ثبات CX را ۶ (تعداد بایت‌های مصرفی ۳ دستور اسمبلی) قراردادهایم و در دستور بعدی، ۳ دستور فوق را در فایل test.com ذخیره کرده‌ایم.

۱۱-۱-۱-۱۶ دستور U (Unassemble)

این دستور موجب می‌شود تا بتوانیم برنامه به زبان ماشین را که در فایل‌های COM یا EXE ذخیره‌شده است به برنامه زبان اسمبلی تبدیل کنیم. این دستور به‌صورت زیر به کار می‌رود:

```
U     [محدوده حافظه]
```

محدوده حافظه، مشخص‌کننده قسمت‌هایی از حافظه است که دستورات زبان ماشین موجود در این بازه، باید به زبان اسمبلی تبدیل شود. اگر دستور U را بدون ذکر محدوده حافظه اجرا کنیم، بر روی 20h بایت عمل خواهد کرد.

مثال)

```
A > debug    TEST.COM
_U   100,L,6
40EB:0100   CD05   INT     05
40EB:0102   B400   MOV     AH,00
40EB:0104   CD21   INT     21
```

به‌راحتی می‌توانیم این برنامه را تغییر دهیم. فرض کنید می‌خواهیم اولین دستور برنامه به INT 18 تبدیل شود:

```
_u    105,L,6
41EB:0100
40EB:0102
40EB:0104
_A    100
40EB:0100 INT       18
40EB:0102
_U    100,L,6
40EB:CD18 INT       18
40EB B400   MOV     AH,00
40EB:CD21 INT       21
_W
Writing       0006    bytes
_Q
A >
```

۱۱-۱-۱-۱۷ دستور C (Compare)

این دستور برای مقایسه محتویات دو قسمت مختلف از حافظه به‌کاررفته و به‌صورت زیر استفاده می‌شود:

C <محدوده حافظه> <آدرس>

دستور C محتویات محل‌هایی از حافظه را که توسط "محدوده حافظه" مشخص می‌شود با همان مقدار از حافظه که از آدرس مشخص‌شده در این دستور شروع می‌شود، مقایسه می‌کند.

مثال)

```
_C    100,1FF         300
```

دستور فوق موجب می‌شود تا محل‌هایی از حافظه که بین آدرس‌های 100 و 1FE است با محل‌هایی از حافظه که از محل 300 به بعد قرار دارد مقایسه شود. بدیهی است که قسمتی از حافظه که قسمتی از حافظه که از محل ۳۰۰ به بعد قرار دارد و باید در مقایسه شرکت کند، برابر با قسمتی از حافظه است که از آدرس 100 تا 1FF قرار دارد. دستور فوق معادل دستور زیر است:

```
_C    100,L,100       300
```

۱۱-۱-۱-۱۸ دستور T (Trace)

این دستور، برنامه موجود در حافظه را به‌صورت دستور به دستور اجرا می‌کند. یعنی پس از اجرای هر دستور، محتویات جدید ثبات‌ها را نمایش می‌دهد و برای اجرای دستور بعدی برنامه، باید یک‌بار دیگر این دستور را صادر کرد. این دستور به‌صورت زیر به کار می‌رود:

T [= آدرس] [مقدار]

آدرس مشخص‌کننده محلی از حافظه است که دستورات موجود در حافظه، از آن محل به بعد باید توسط دستور T اجرا شوند. <مقدار>، تعداد دستوراتی را مشخص می‌کند که دستور T باید آن‌ها را اجرا کند، اگر منظور نشود، یک فرض خواهد شد.

مثال)

```
_T = 011H    10
```

۱۱-۱-۱-۱۹ دستور P (Proceed)

این دستور برای فراخوانی زیر برنامه‌ها، اجرای وقفه‌های نرم‌افزاری و اجرای دستورات داخل حلقه تکرار مورداستفاده قرار می‌گیرد و به‌صورت زیر به کار می‌رود:

```
P [ = آدرس] [مقدار]
```

آدرس، مشخص‌کننده محل وجود دستوری است که باید اجرا شود و اگر آدرس مشخص نشود، دستوری که آدرس آن در ثبات IP است منظور خواهد شد. مقدار، تعداد دستوراتی را مشخص می‌کند که باید اجرا شوند. اگر ذکر نگردد، یک فرض خواهد شد.

اگر اولین دستوری که باید توسط IP اجرا شود، شروع یک حلقه تکرار، وقفه نرم‌افزاری، دستور تکرار روی رشته‌ها و یا فراخوانی زیر روال نباشد، این دستور با دستور T یکسان است. دستور P محتویات ثبات‌ها را در خاتمه کار در صفحه‌نمایش، نشان می‌دهد.

مثال) این دستور موجب اجرای دستور موجود در محل 100H می‌شود.

```
_P = 100
```

۱۱-۱-۲ پیام‌های خطای Debug

در حین کار با برنامه debug چنانچه دچار اشتباهی شده باشیم، پیام‌های خطایی صادر می‌شود که در زیر موردبررسی قرار می‌گیرند.

خطای BF : در صورتی بروز می‌کند که در حین اجرای فرمان G (go) از بیش از ۱۰ نقطه توقف استفاده کرده باشیم.

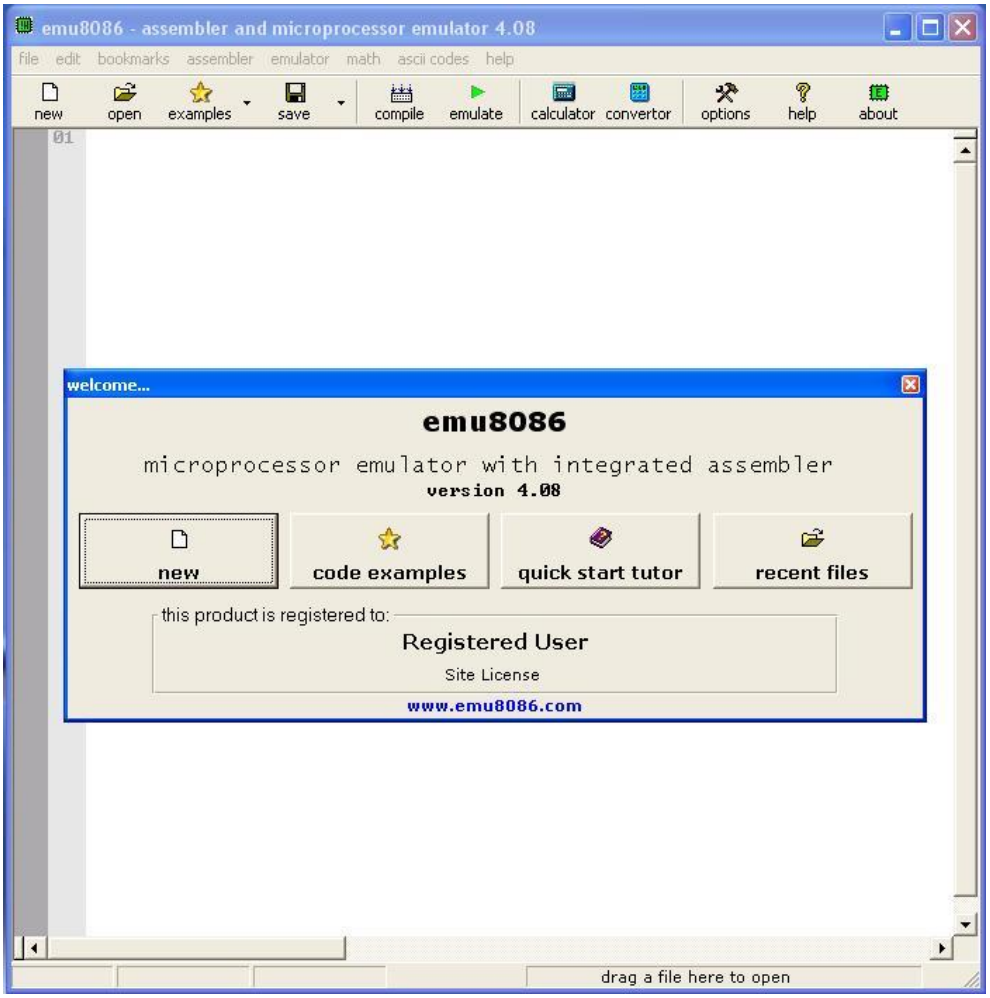
خطای BP : در صورتی بروز می‌کند که خواسته باشیم یک فلگ (flag) را مقدار بدهیم ولی از کاراکتر غیرمجازی استفاده کرده باشیم.

خطای BR : در صورتی بروز می‌کند که حین استفاده از فرمان R (register) به‌جای نام ثبات از یک نام غیرمجاز استفاده کرده باشیم.

خطای DF : در صورتی بروز می‌کند که برای یک نشانگر (flag) دو مقدار وارد کرده باشیم.

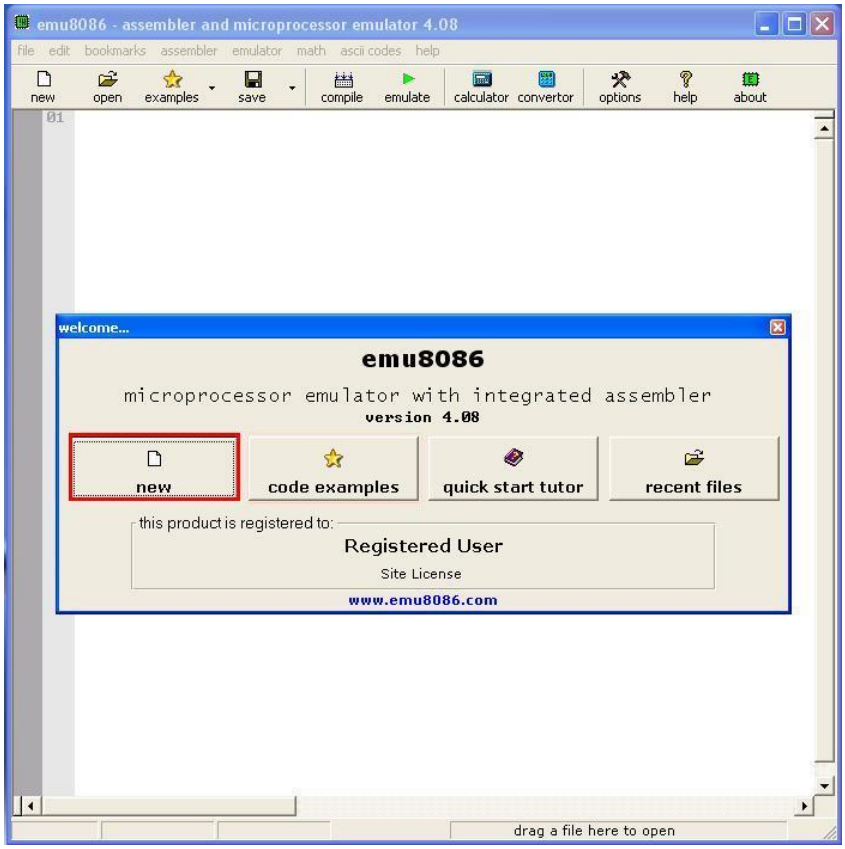
۱۱-۲ آشنایی با نرم‌افزار EMU8086

Emu8086 یک نرم‌افزار اسمبلر است که برای تبدیل کدهای اسمبلی میکروپروسسور ۸۰۸۶ به کد ماشین طراحی‌شده است. در این نرم‌افزار قابلیت مشاهده وضعیت حافظه و ثبات‌های میکرو ۸۰۸۶ وجود دارد . علاوه بر این وجود امکانات جانبی در این نرم‌افزار کار را برای کاربر آسان کرده است .

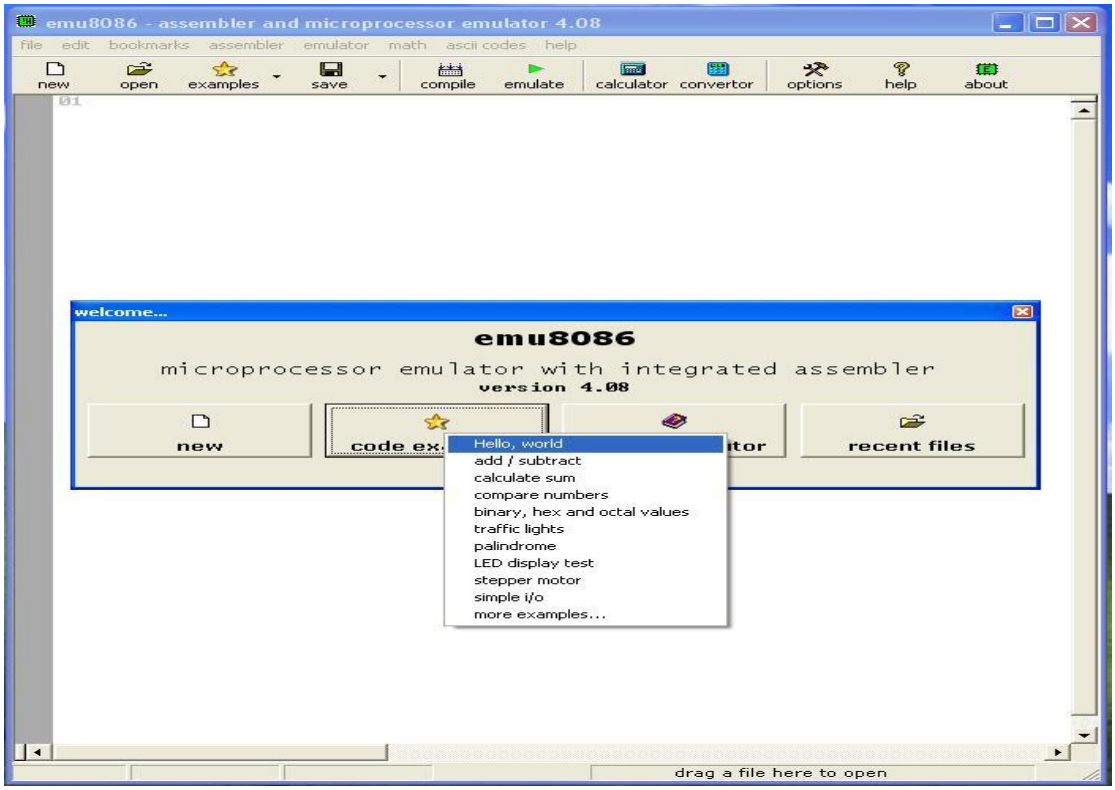


مراحل نوشتن برنامه و اجرای آن با استفاده از emu8086

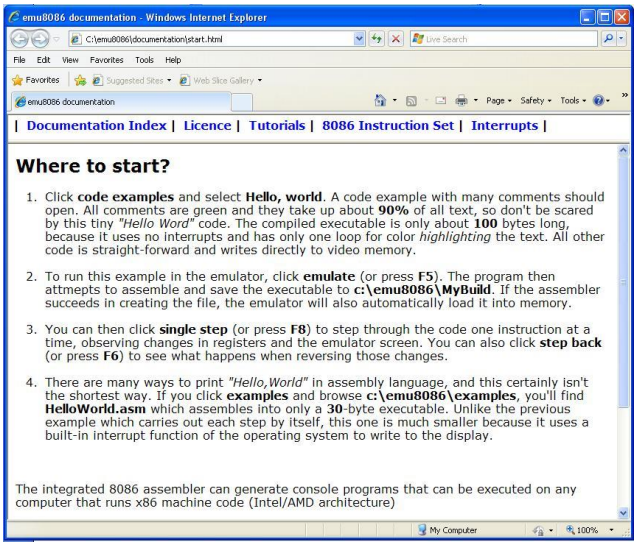
۱۱-۲-۱ ایجاد یک پروژه جدید



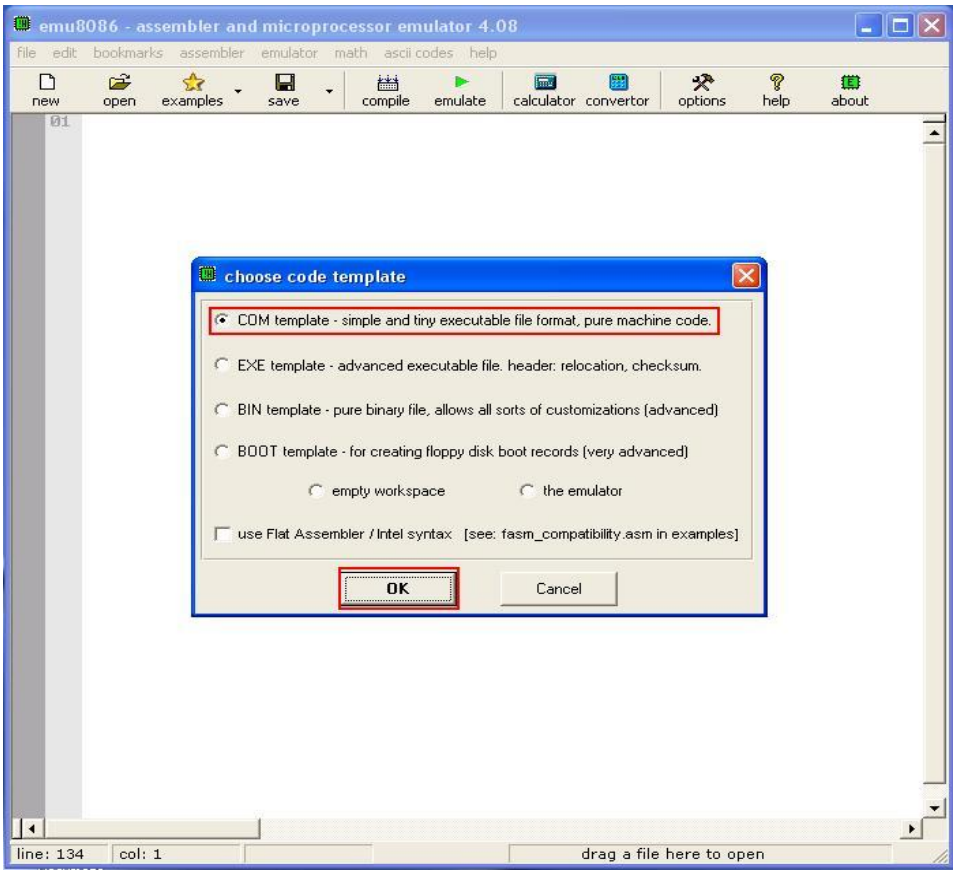
معرفی گزینه‌های در صفحه اول: با استفاده از گزینه Code Example می‌توانید به نمونه برنامه‌های اسمبلی که برای نمونه آورده شده است دسترسی داشته و آن‌ها را اجرا کنید.



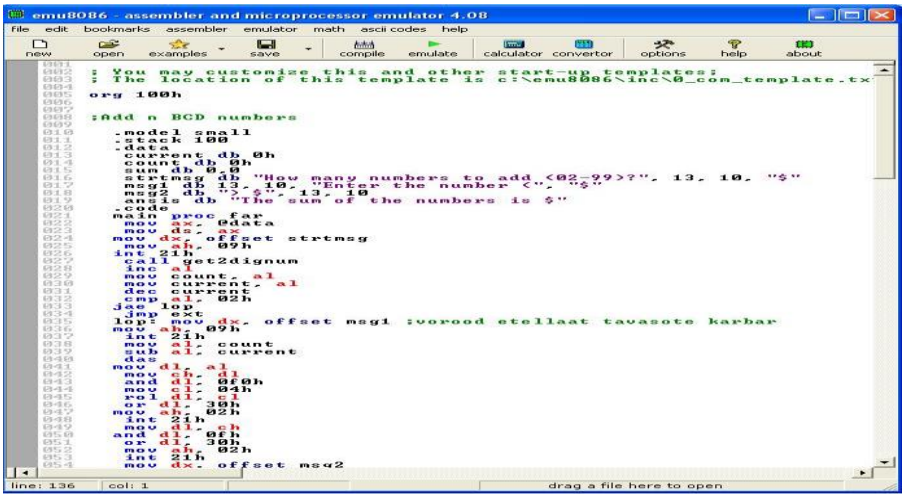
با زدن گزینه quick start tutor صفحه شکل زیر باز می‌شود که در آن راهنمایی سریعی برای استفاده از emu8086 وجود دارد .



با استفاده از گزینه recent files می‌توانید به آخرین برنامه‌های شبیه‌سازی‌شده توسط emu8086 دسترسی پیدا کنید. برای ایجاد یک پروژه جدید پس از باز کردن برنامه بر روی گزینه new کلیک کنید. با انجام این کار پنجره‌ای باز می‌شود که هرکدام از گزینه‌ها را می‌توان انتخاب کرد ولی انتخاب گزینه‌ی اول و زدن OK چون فقط کد ماشین تولید می‌کند برای شروع مناسب‌تر است . چون گزینه‌های دیگر برای کارهای پیشرفته‌تر است و کار با آن‌ها کمی مشکل است . البته زدن گزینه cancel هم مشکلی در اجرای برنامه ایجاد نمی‌کند.



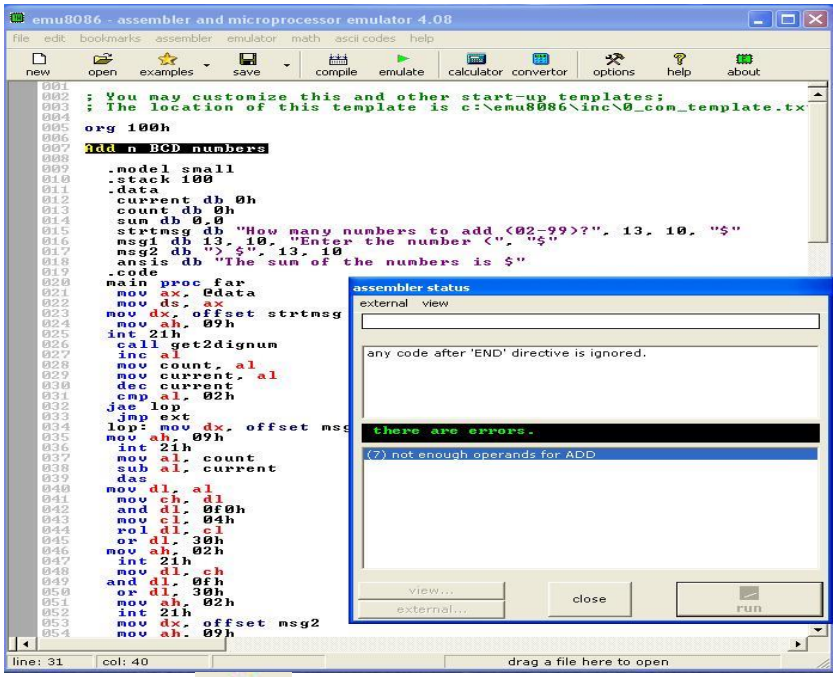
پس از زدن OK صفحه ویرایشگر متن‌باز می‌شود که کد برنامه اسمبلی را در جای مشخص‌شده می‌نویسیم.

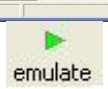


پس از نوشتن برنامه می‌توانیم آن را کامپایل کرده و خطاهای احتمالی در برنامه را برطرف کنیم .

برای کامپایل کردن برنامه بر روی آیکون 

کلیک کرده و در صورت وجود خطابه شما اخطار می‌دهد و محل وجود خطا را نیز با مشکی کردن آن خط نشان می‌دهد.



پس از کامپایل کردن و برطرف کردن خطا با کلیک بر روی آیکون 

می‌توان برنامه را شبیه‌سازی کرد

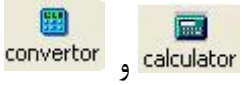
معرفی آیکون‌های صفحه emu8086 –assembler : با کلیک بر روی آیکون 

می‌توانید پروژه جدیدی بازکنید و با استفاده از 

می‌توانید به فایل‌هایی که قبلاً اجرا کرده و save کرده‌اید را بارگذاری کنید . با استفاده از گزینه 

به مثال‌های آماده‌ای از برنامه‌نویسی اسمبلی برای میکرو ۸۰۸۶ نوشته‌شده دسترسی داشته باشید . 

با کلیک بر روی این می‌توانید پروژه خود را saveکنید . گزینه‌های

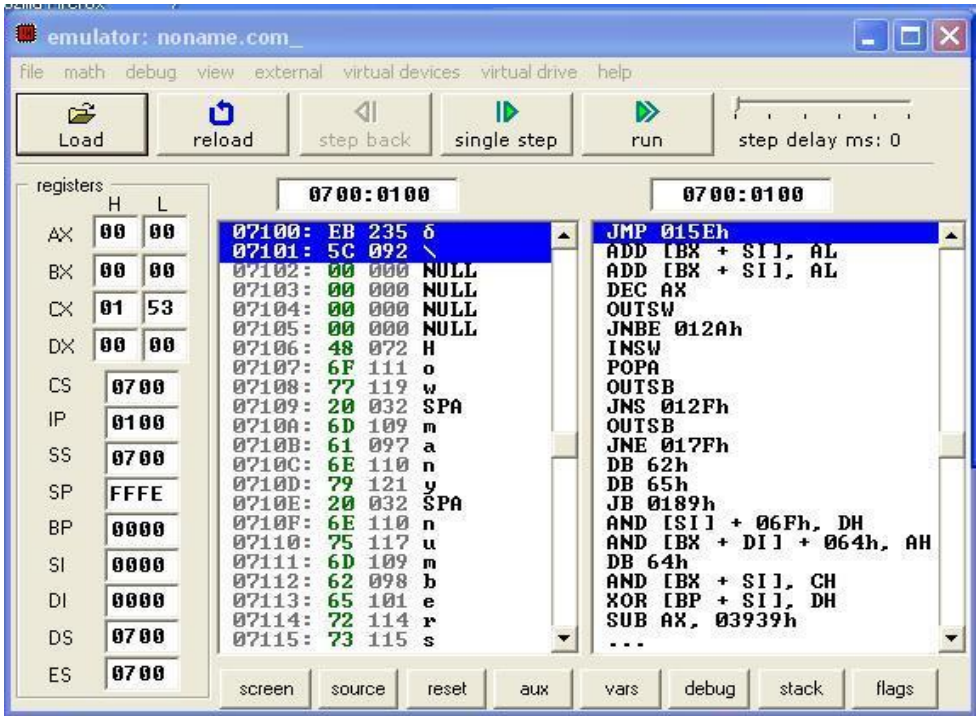


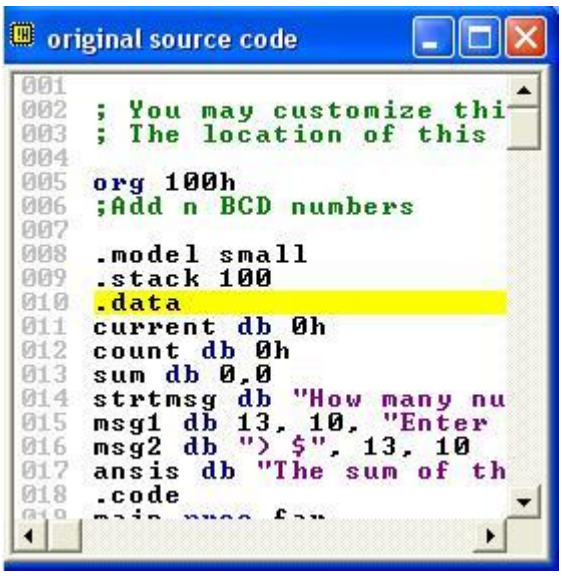
و امکانات مناسبی برای انجام محاسبات ریاضی و تغییر سیستم‌های اعداد مثل باینری به هگزا و ... استفاده کرد . با کمک 

می‌توانید نحوه استفاده از نرم‌افزار و همچنین راهنمایی‌هایی برای نوشتن برنامه اسمبلی برای میکرو ۸۰۸۶ آشنا شوید.

گفتیم که با کلیک بر روی 

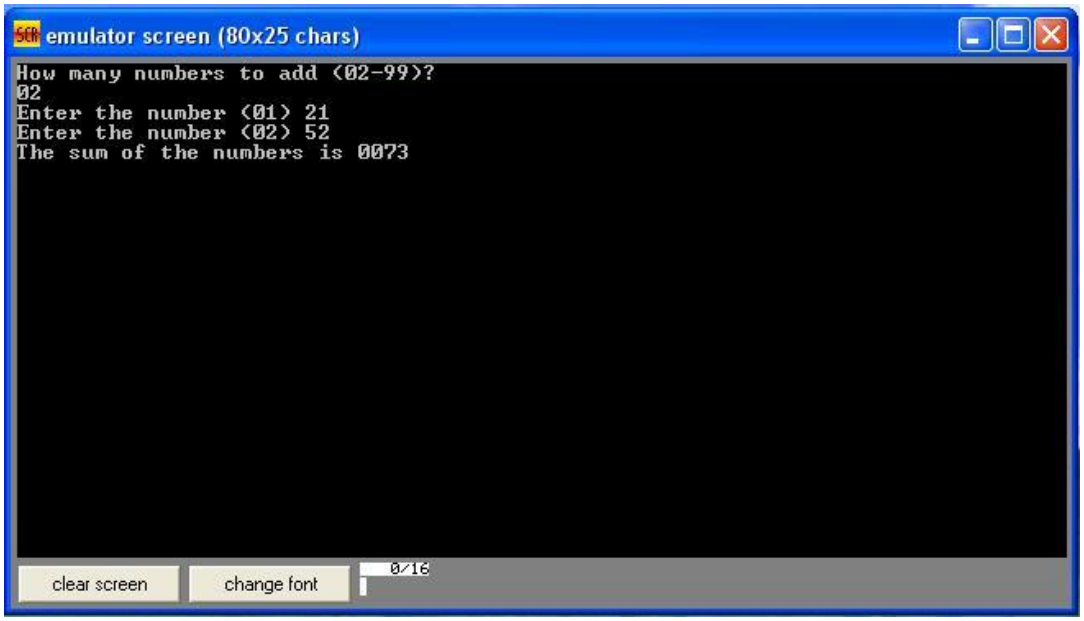
می‌توان برنامه را شبیه‌سازی کرد . با این کار دو صفحه جدید باز می‌شود یکی صفحه emulator و دیگری صفحه original source code (شکل۴-۹)



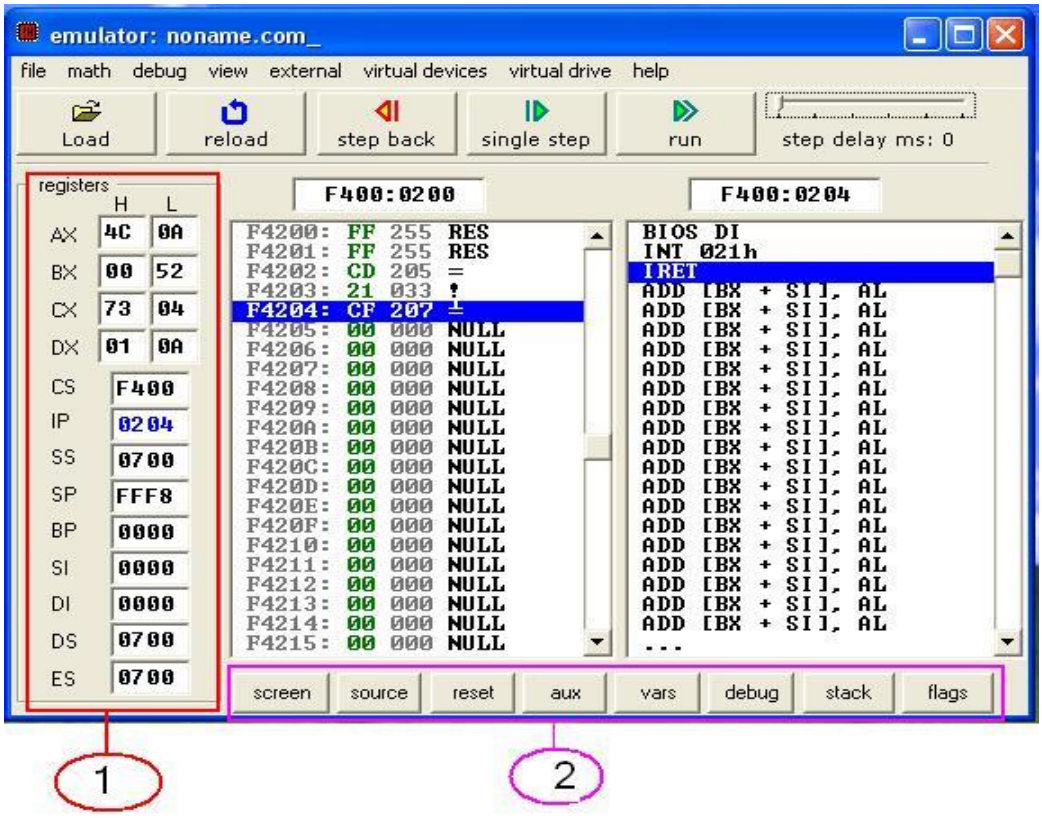


برای شبیه‌سازی بر روی آیکون 

در صفحه emulator کلیک کنید تا برنامه اجرا شود . در صورت اجرا صفحه emulator screen (شکل۴-۱۰)باز می‌شود و که در آن خروجی‌های برنامه را می‌توانید مشاهده کنید .



پس از اجرای برنامه در قسمت سمت چپ صفحه emulator مقدار هر یک از رجیستر ها پس از اجرای برنامه و یا در حین اجرا را می‌توانید ببینید . و با کلیک بر روی هر یک از آیکون‌های پایین صفحه emulator می‌توانید مقدار stackوvars ... را ببینید .



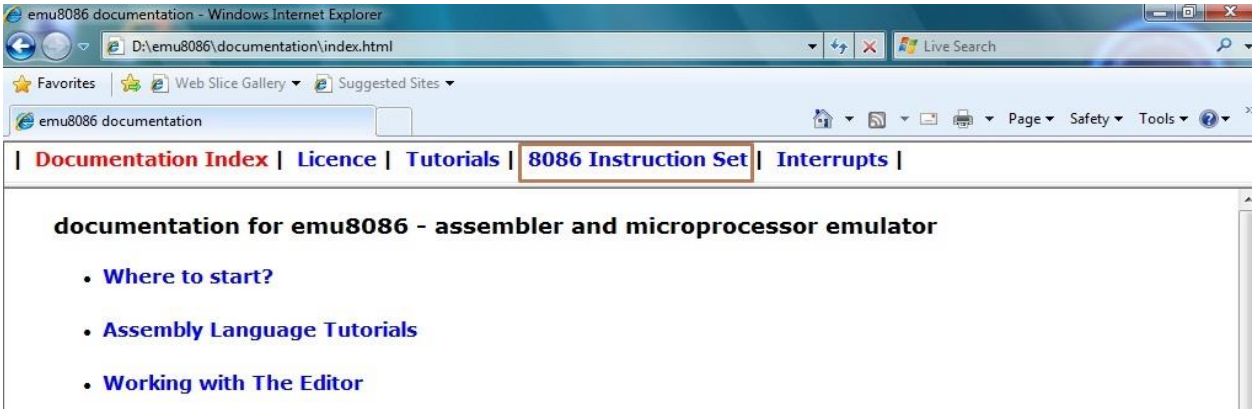
می‌توان با استفاده از گزینه 

می‌توان برنامه را خط به خط اجرا کرد و با استفاده از گزینه 

می‌توان به خطوط قبلی برنامه برگشت . مزیت این کار این است که عملکرد هر خط برنامه را می‌توان مشاهده کرد و درصورتی‌که اشکالی در برنامه باشد می‌توان آن را برطرف کرد.

برای دسترسی به Instruction set میکرو ۸۰۸۶ بر روی آیکون Help 

کلیک کرده و در صفحه بازشده بر روی 8086 Instruction set (که با کادر قرمزرنگ مشخص‌شده است) کلیک کنید در صفحه بازشده تمامی دستورات ۸۰۸۶ قابل‌مشاهده است .



منابع

۱- ع. جعفر نژاد قمی، ۱۳۸۲، برنامه‌نویسی به زبان اسمبلی، نشر علوم رایانه، صفحه.

۲- ح. سید رضی، ۱۳۸۵، زبان ماشین و اسمبلی و کاربرد آن در کامپیوترهای شخصی، انتشارات ناقوس – زانیس، صفحه.

۳- ف. عبدالهی، ۱۳۸۷، آموزش برنامه‌نویسی به زبان اسمبلی، انتشارات دانشگاه آزاد واحد نجف‌آباد، صفحه.

۴- ح. مقسمی، ۱۳۸۴، زبان ماشین و اسمبلی، انتشارات گسترش علوم پایه، صفحه.

۵- م. مزیدی، ۱۳۸۸، برنامه‌نویسی به زبان اسمبلی در کامپیوترهای IBM 80x86 و سازگار با آن، انتشارات باغانی، ۳۶۹ صفحه.

۶- سایت www.mohandesyar.com