سعید دانشگر

به نام خدا

ASM — کمبود Seglaze

منابع : Digital Design مانو . فصل ۸

RTL

Computer system Architecture مانو فصول ۴، ۵، ۶، ۷، ۸ و قسمتی از ۱۱

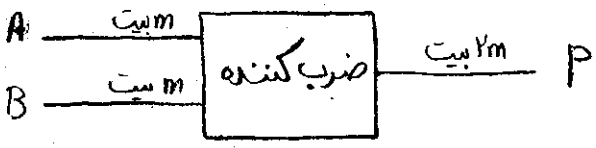
ارزیابی : ۱۷ نمره ← میان‌ترم ۳۰٪ - ۴۰٪ ← شرط قبولی ۸،۵

پایان‌ترم ۶۰٪ - ۷۰٪

۳ نمره ← تکلیف و کوئیز ← ۱،۵

ASM chart (Algoritmic state Machine):

مثال :



اگر خروجی مدار فقط تابع ورودی‌های مدار باشد می‌توان آن را یک مدار ترکیبی یا ماشین مور

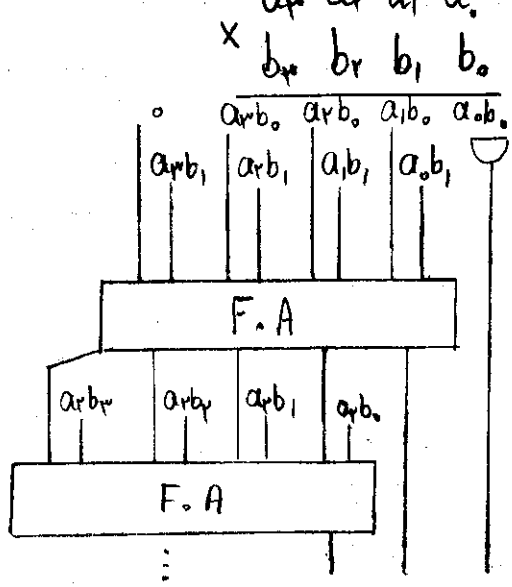
راه‌حل‌ها : ۱- راه‌حل کلاسیک .

۲- استفاده از یک ROM .

۳- جمع و شیفت ترکیبی .

۴- جمع و شیفت ترتیبی

۵- جمع و شمارش ترتیبی .



برای ضرب $n \times n$ نیاز به n^2 گیت AND ، $n-1$ عدد F.A داریم.

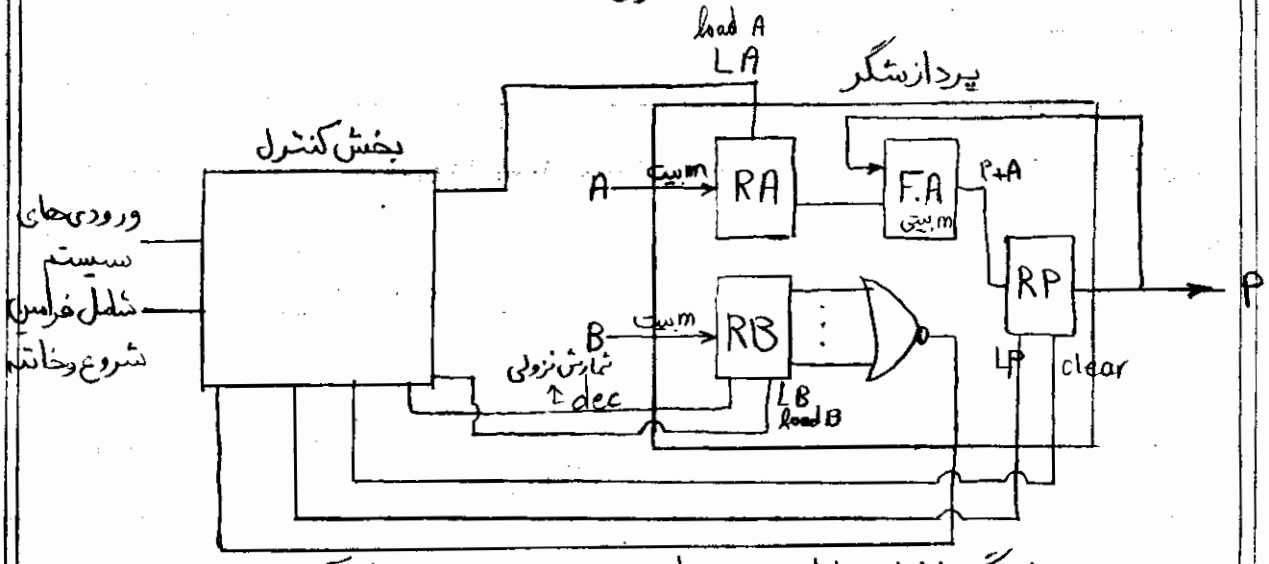
Asm chart ، RTL ابزارهای طرح مدار ترتیبی بزرگ سنکرون کلاک مد هستند.

دستورات کنترلی: دستوراتی هستند که می‌توانند ترتیب و توالی مدار را به هم بزنند. این دستورات توالی را مشروط و یا غیرمشروط به هم می‌زنند.

```
        ← Read A, B
        ← P=0
        10 if B=0 then stop
        ← P=P+A
        ← B=B-1
        goto 10
```

جملات پردازشی

دستورات کنترلی



بخش کنترل شامل دستورات کنترلی

پردازشگر شامل مدارات دستورات غیرکنترلی

ترتیب اجرا:

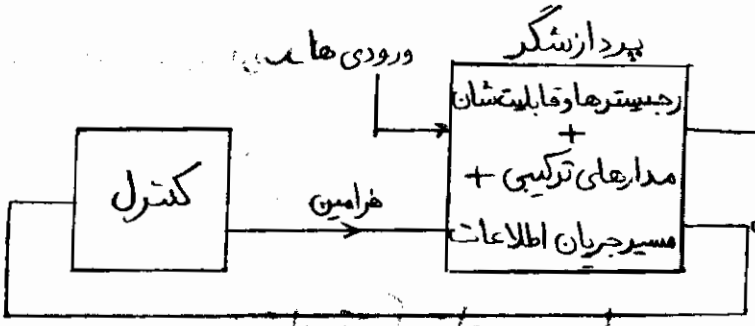
۱. نقطه شروع را قرار داد می‌کنیم.

۲. توالی را قرار داد می‌کنیم.

۳. دستورات کنترلی

LA	LB	LP	clear	Dec	
1	1	0	0	0	← در کلاک اول
0	0	0	1	0	⋮
0	0	1	0	0	
0	0	0	0	1	

از دید مداری :

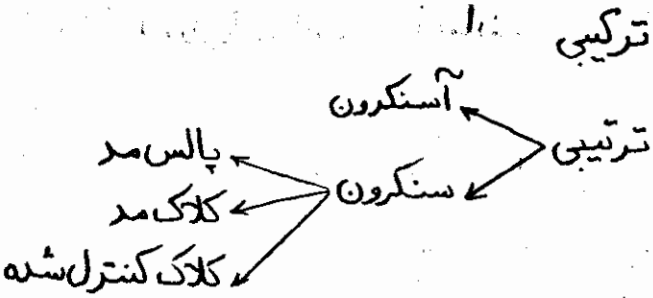


مدار بالا، مدار ترتیبی سنکرون کلاک مد است.

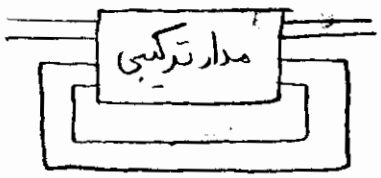
همان طور که دیدیم در بخش کنترل ترتیب اجرا را پیاده سازی می کنیم که شامل :

۱- نقطهٔ شروع (قراردادی) ۲- توالی (قراردادی) ۳- جملات کنترلی است

ASM را فقط برای مدار ترتیبی سنکرون کلاک مد می نویسیم.

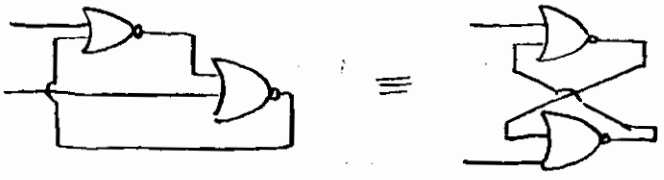


آنچه که به عنوان مدار آسنکرون بحث می شود در شکل



مقابل نشان داده شده است:

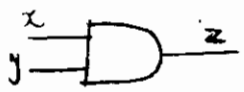
به عنوان مثال فلیپ- فلاپ



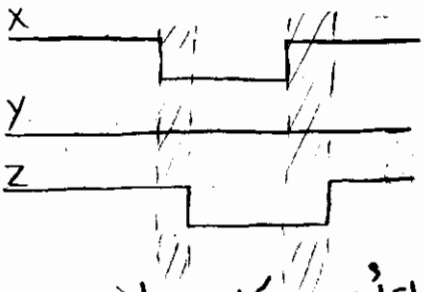
یک مدار آسنکرون است :

مشکلات مدار آسنکرون عبارتند از : ۱- Cycle ۲- Race ۳- Hazard

همهٔ مدارهای ترکیبی تأخیر در خروجی دارند:



و تنها راه حل آن این است که از وقتی که تمام ورودی ها



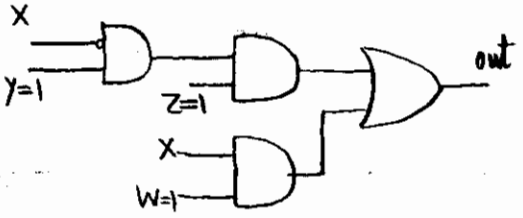
وارد مدار شدند باید به اندازهٔ ماکزیمم تأخیر صبر کنیم..

حال در مدار آسنکرون چون فیدبک وجود دارد همراه با تأخیر مشکلات زیادی بوجود می آید.

مشکل Race : در مداری که چند خروجی دارد چون خروجی ها از مسیرهای متفاوتی تشکیل شده اند لذا با تغییر ورودی همهٔ خروجی ها همزمان تغییر حالت نمی دهند (به علت تأخیر). حال اگر این خروجی ها فیدبک شده باشند مدار وارد حالت دلخواه ما نمی رود. (مسابقه بین خطوطی که فیدبک شده اند)

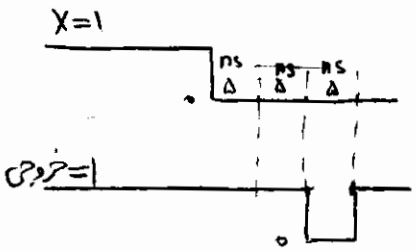
مشکل cycle : حالتی است که با تغییر ورودی، مدار به نوسان بیفتد. مانند فلیپ فلاپ RS که ورودی آن از R=1 , S=1 به R=0 , S=0 برود.

مشکل Hazard : مانند Race است ولی در یک خط اتفاق می افتد. در مدار زیر با تغییر X خروجی بعد از چند تغییر حالت، پایدار می شود



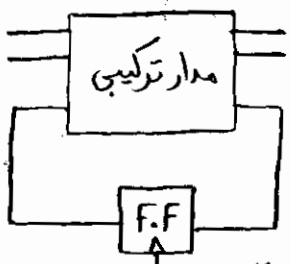
که مشکل Hazard را بیان می کند.

به عنوان مثال اگر تمام گیت‌های شکل فوق



دارای تأخیر 5ns باشند آن گاه :

مدار سنکرون در شکل مقابل نشان داده شده است:

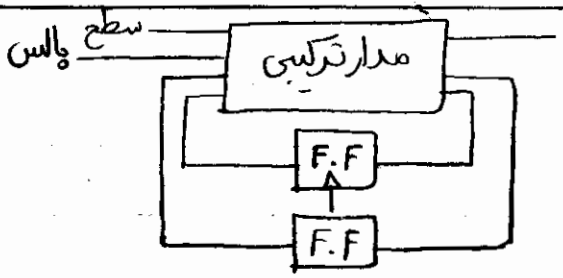


درمدار مقابل اگر عرض کلاک پالس بسیار کوتاه باشد مشکلات قبل راحل می کند اما این راه حل عملی نیست. لذا اگر از فلیپ فلاپ حساس به لبه یا Master Slave استفاده کنیم درعمل نیز مشکلی نخواهیم داشت.

درمدار سنکرون کلاک مد، کلاک به طور مستقیم به همهٔ فلیپ فلاپ ها متصل است. اگر واسطه ای درمیان باشد مدار سنکرون با کلاک کنترل شده ماست.

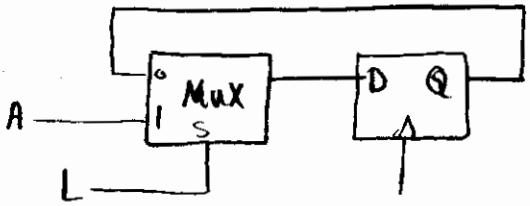
آنچه که به عنوان یک ورودی برای مدار درنظر گرفته می‌شود سطح نام دارد و ۱ یا ۰ آن معنی دارد. امّا آنچه که به عنوان کلاک درنظر گرفته می‌شود از نوع پالس است یعنی ۱ یا ۰ آن اهمیتی ندارد بلکه تغییر حالت آن مهم است.

در مدار پالس مد سنج پالس واحد نیست و فقط به کلاک نمی آید بلکه به ورودی هایی تواند بیاید و فلیپ فلاپ هامی توانند کلاک داشته باشند یا نداشته باشند :

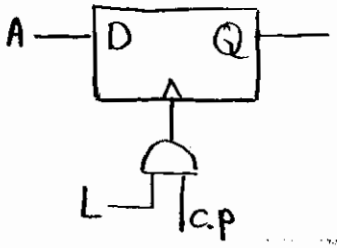


مثال: یک D فلیپ فلاپ داریم می‌خواهیم با فرمان load مقدار ورودی A در فلیپ فلاپ قرار گیرد.

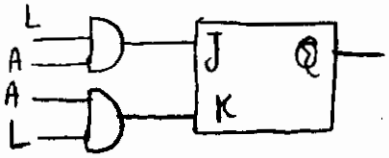
حل به فرم کلاک مد:



حل در حالت کلاک کنترل شده:



اگر فلیپ فلاپ JK بود به شکل مقابل عمل می‌کردیم:

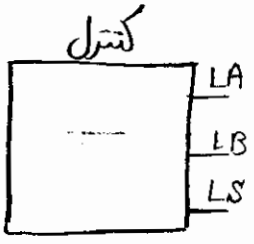


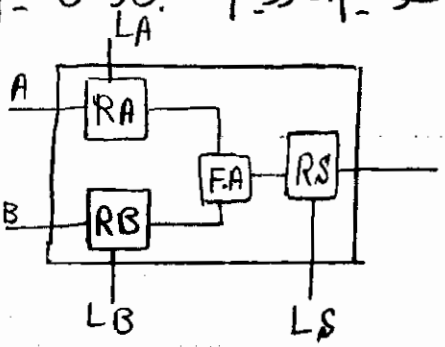
clear سنکرون ← clear ای است که با کلاک AND می‌شود

clear آسنکرون ← clear ای که مستقل از کلاک و... مدار را clear می‌کند.

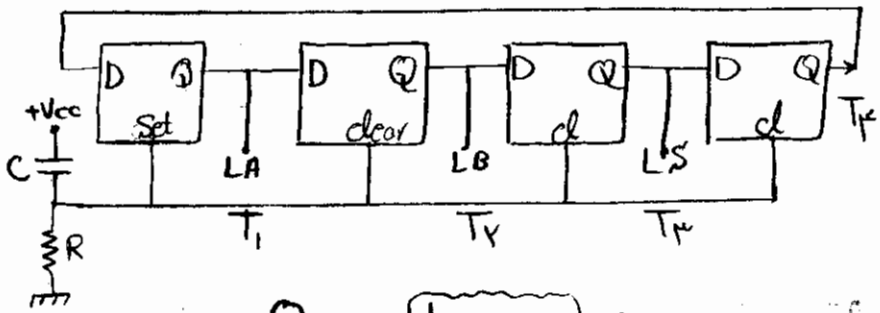
حال می‌خواهیم الگوریتم مقابل را حل کنیم:

```
10  Read A
    Read B
    S=A+B
    Goto 10
```





Ring Counter



LA	LB	LS
1	0	0
0	1	0
0	0	1
0	0	0

Q :

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

در مدار بالا حداقل پریود برای اینکه جمع کننده بتواند عمل کند باید D باشد که D تأخیر گیت F.A است.

چون هیچ شرطی نداشتیم لذا از واحد پردازش هیچ خروجی به واحد کنترل نرفته است.

μ-OP : عملی است که برای انجام آن یک لبهٔ کلاک لازم است. این عمل بر روی محتوای یک یا چند رجیستر انجام می‌شود و نتیجه در یکی از همان رجیسترها یا رجیستر دیگر ذخیره می‌شود.

10	Read A	$RA \xleftarrow{\mu\text{-OP}} A$
	Read B	$RB \leftarrow B$
	$S = A + B$	$RS \leftarrow RA + RB$
	goto 10	

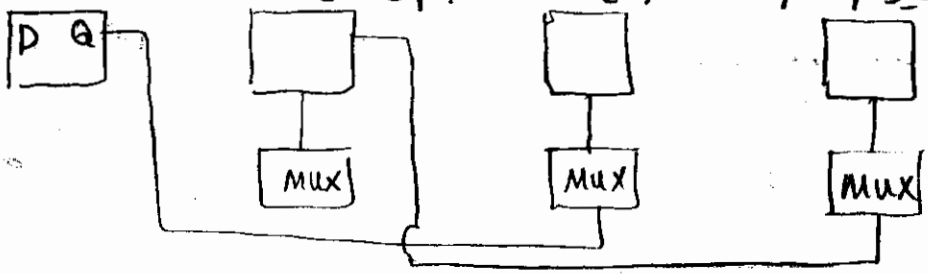
در مدار فوق جمع‌کننده کلاک ندارد و در مورد پریود کلاک برای عمل کردن جمع کننده بحث شد و به طور کلی رجیسترها و فلیپ فلاپ‌ها کلاک نیاز دارند.

در تعریف μ-op باید دقت داشت که یک لبه کلاک برای انجام کار نیاز است ولی

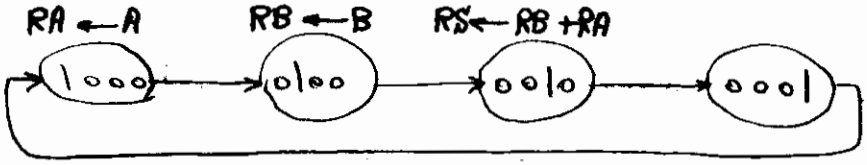
اهمیتی ندارد که پریود کلاک چقدر باشد. ممکن است عملی که با دو لبه انجام می‌شود با یک کلاک طولانی‌تر انجام شود.

شیفت دادن یک رجیستر به اندازهٔ دو واحد می‌تواند μ-op باشد یا نه.

درحالت زیر μ-op است که با یک لبه کلاک انجام می‌شود:

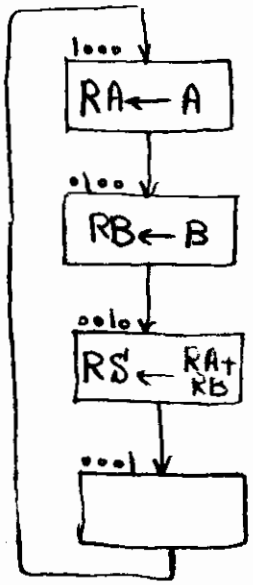


در مثالی که مطرح شد مدار کنترل دارای ۴ حالت بود که نمودار آن در زیر است:



زمان اجرای μ-op: هرحالت واحد کنترل تعیین کننده زمان انجام μ-op است

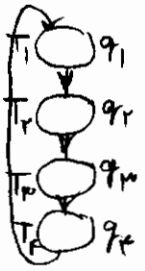
ASM چارت مدار فوق به شکل زیر می‌شود:



μ-op ها
RA ← A
RB ← B
RS ← RA + RB
بخش پردازشگر

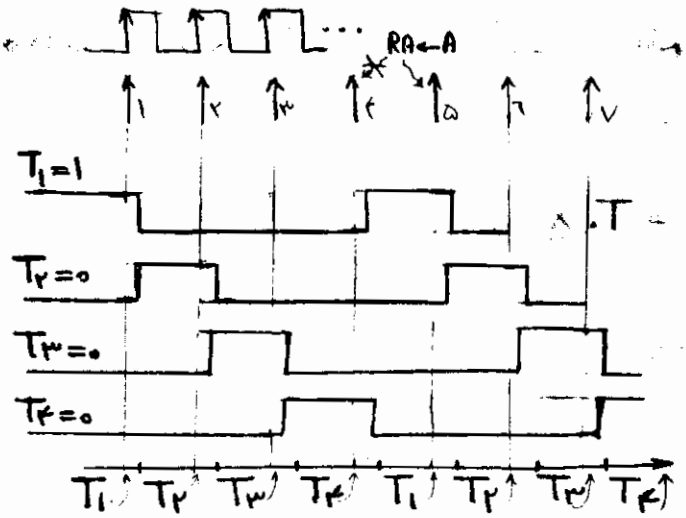
μ-op ها را از روی ASM چارت بدست می‌آوریم

نمودار حالت واحد کنترل را بدست می‌آوریم

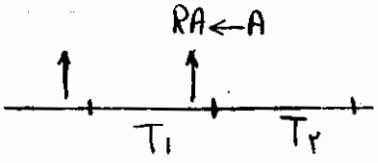


یک μ-op در یکی از حالتهای واحد کنترل و با یک لبهٔ کلاک انجام می‌شود.

یک کلاک باعث انجام μ-op ها در واحد پردازشگر و تغییر حالت‌ها در واحد کنترل می‌شود.

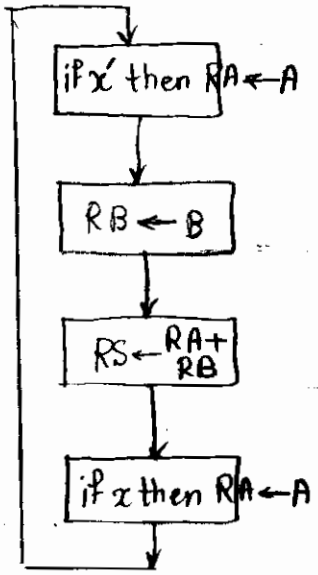


با احتساب تأخیرها می‌بینیم که کلاک شمارهٔ ۵ است که باعث بارگیری رجیستر A می‌شود نه کلاک شمارهٔ ۴. لذا کلاکی باعث انجام μ-op های حالت T_1 می‌شود که مارا از حالت T_1 به T_2 می‌برد.



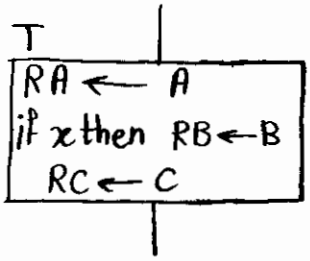
μ-op هایی که تاکنون دیدیم غیرمشروط بودند.

نمونه ای از μ-op مشروط در زیر آمده است:



X ورودی واحد کنترل است که به‌صورت شرط در چارت ASM ظاهر می‌شود.

در حالت مقابل سه μ-op شامل :

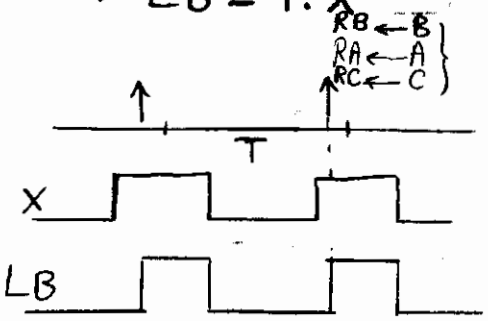


$RA \leftarrow A$, $RB \leftarrow B$, $RC \leftarrow C$

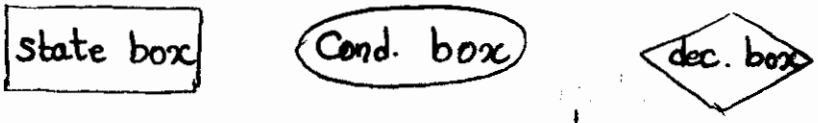
وجود دارند که $RB \leftarrow B$ مشروط و بقیه غیرمشروط هستند. در این حالت،

$$LA = T \quad , \quad LC = T$$

$$, \quad LB = T.X$$

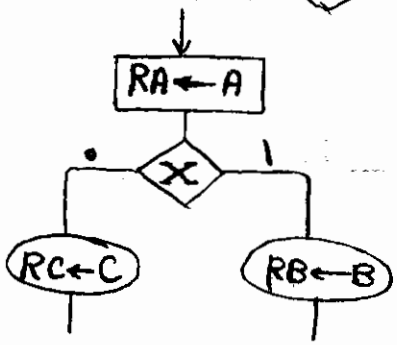


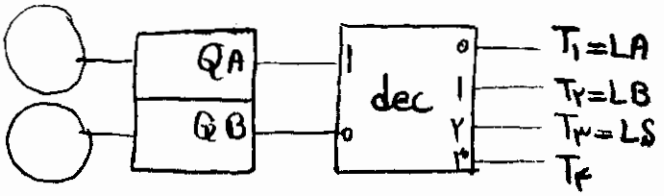
اجزاء ASM چارت :



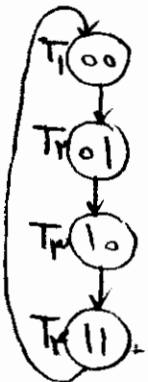
RA ← A
if x then RB ← B
if x' then RC ← C

≡





←

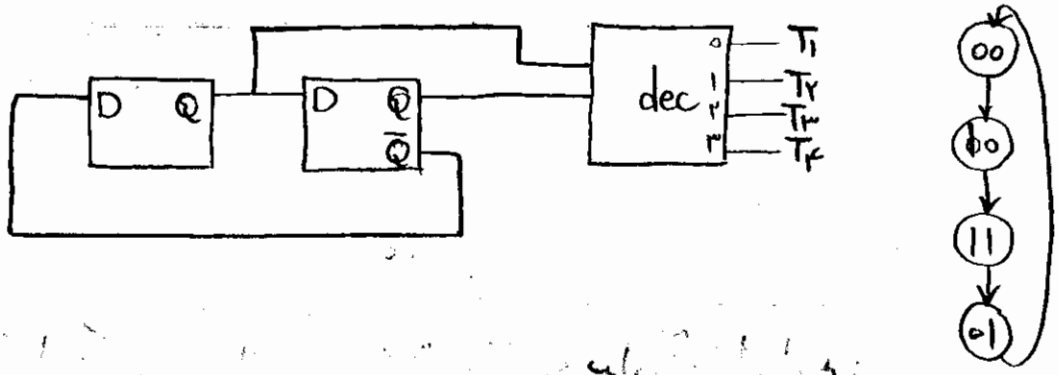


در شکل بالا نمونهٔ دیگری از واحد کنترل برای مسئله مطرح شده ارائه شده است.

Ring Counter

مدار فوق از نوع CD (Counter- decoder) است. در قبل نوع RC را دیدیم.

حال از JC (Johnson Counter) استفاده می کنیم.



جمع بندی روشها:

RC	JC	CD
14 F.F	8 F.F	4 F.F
به تعداد حالتها		
dec ندارد	AND گیت ۱۶ (دو ورودی)	AND گیت ۱۶ (۴ ورودی)

درمورد JC اگر ۳ فلیپ فلاپ داشته باشیم:

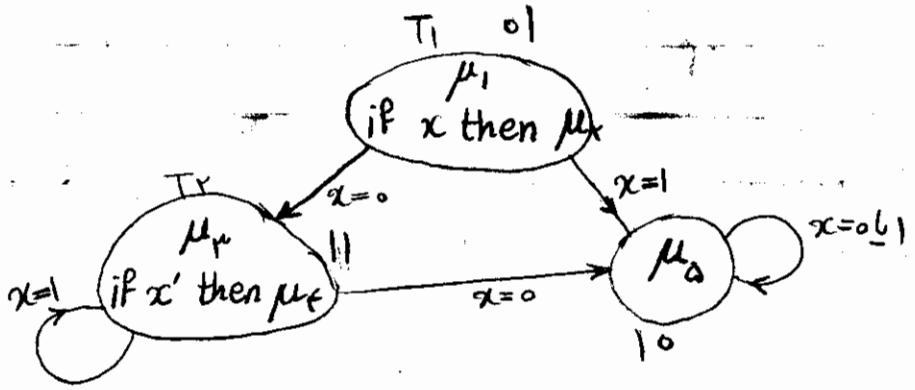
Q_A	Q_B	Q_C	
0	0	0	T_1
1	0	0	$T_2 = Q_A Q'_B$
1	1	0	T_3
1	1	1	T_4
0	1	1	T_5
0	0	1	T_6

نکته: تعداد حالتهای واحد کنترل برابر تعداد state box های ASM چارت است.

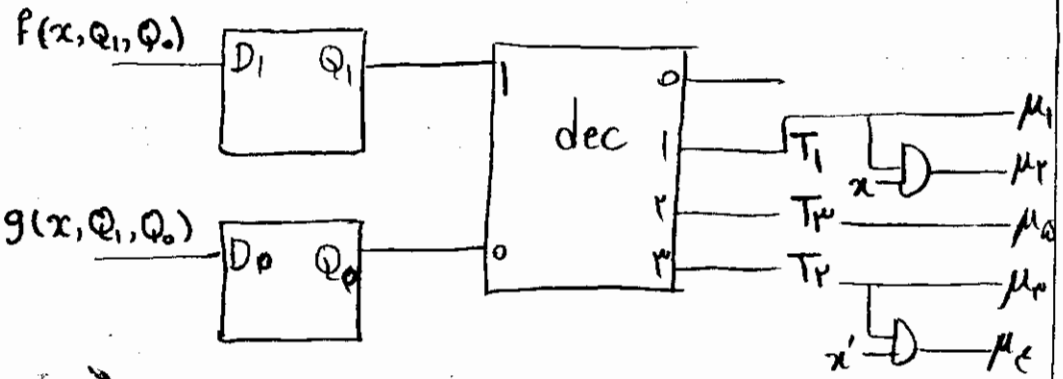
پردازشگر را می توانیم با بدست آوردن لیست μ-op ها از ASM چارت طراحی کنیم.

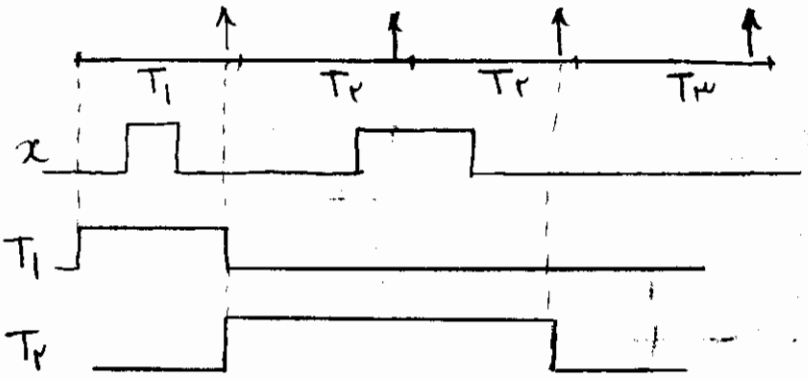
تعداد ورودی های واحد کنترل با تعداد dec box های ASM چارت برابر است.

مثال :

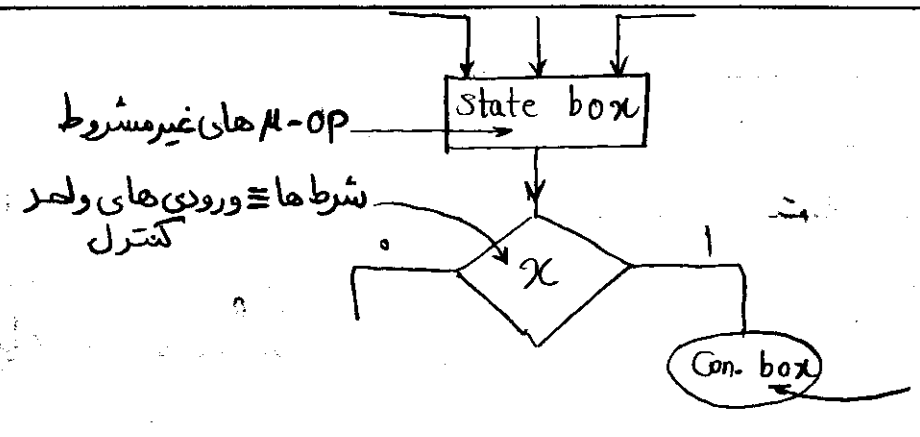


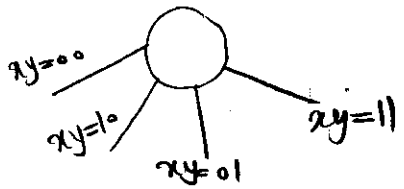
چون واحد کنترل ۳ حالت دارد ۲ فلیپ فلاپ برای آن نیاز داریم :





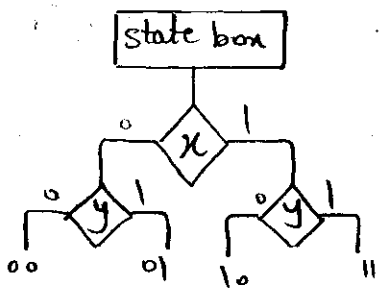
حال ASM چارت مدار بالا را می کشیم :



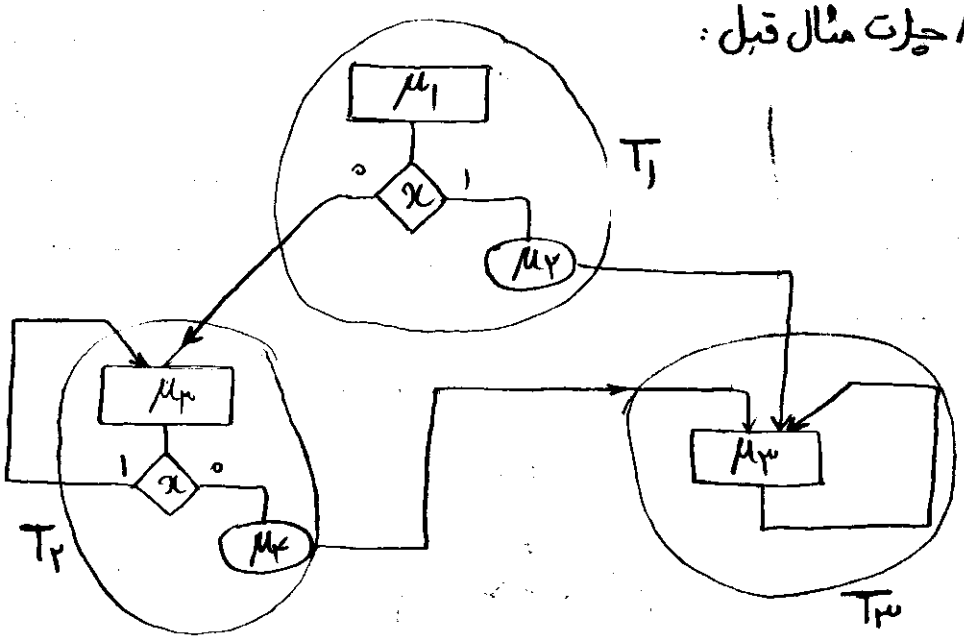


برای نمودار حالت شکل مقابل ASM چارت

زیر را خواهیم داشت:



ASM چارت مثال قبل:



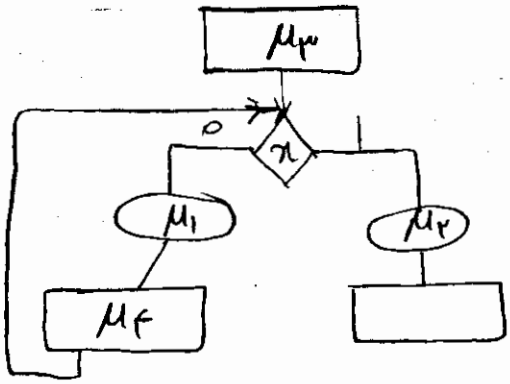
بلوک ASM : یک state box و متعلقات آن را می گوئیم. در شکل بالا هر دایره

یک بلوک است.

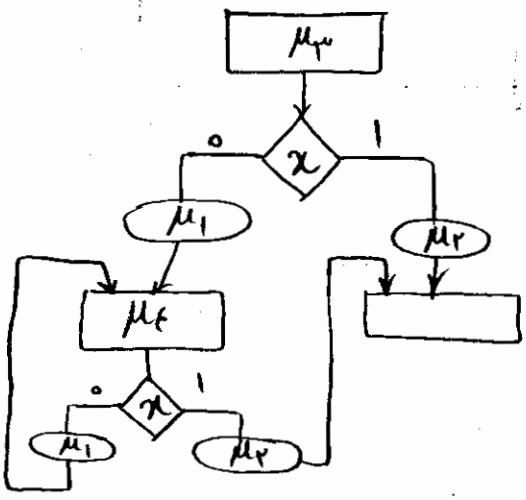
استقلال بلوک ها : بلوک‌ها نمی‌توانند جزء مشترک داشته باشند. یعنی نمی‌توانیم dec box یا Cond. box مشترک داشته باشیم.

بلوک‌ها را با واپرا نشان نمی‌دهیم بلکه از یک state box شروع کرده و مسیر را ادامه می‌دهیم تا به state box بعدی برسیم. آنچه که در این مسیر وجود خواهد داشت اجزای یک بلوک را تشکیل می‌دهد.

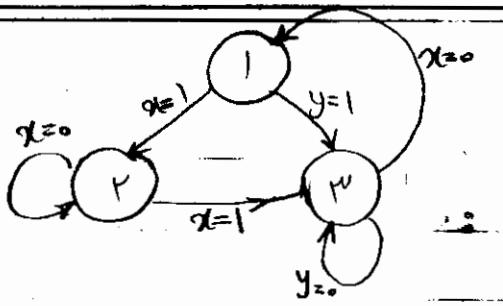
در ASM چارت زیر جزء مشترک وجود دارد و لذا غلط است.



شکل بالا از ساده کردن فلوچارت زیر حاصل شد:

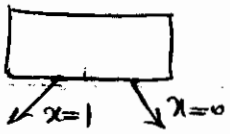


دیاگرام حالت زیر غلط است :

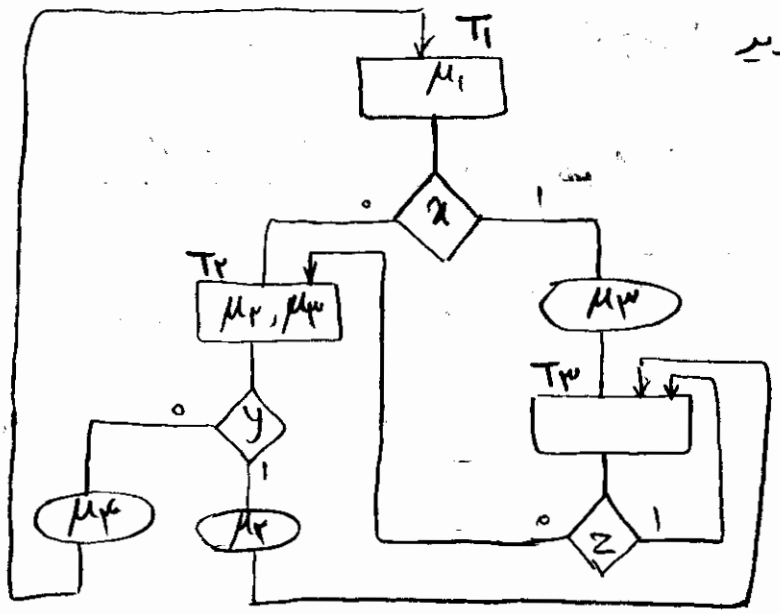


چنین دیاگرام هایی وقتی غلط است که به ازای یک ترکیب معین از x , y مسیر مشخصی وجود نداشته باشد و یا چند مسیر وجود داشته باشد . در شکل بالا در حالت ۳ به ازای $x=0$, $y=0$ دو حالت وجود دارد .

به همین دلیل هیچ گاه در ASM چارت شروط را به شکل زیر نمایش نمی دهیم :



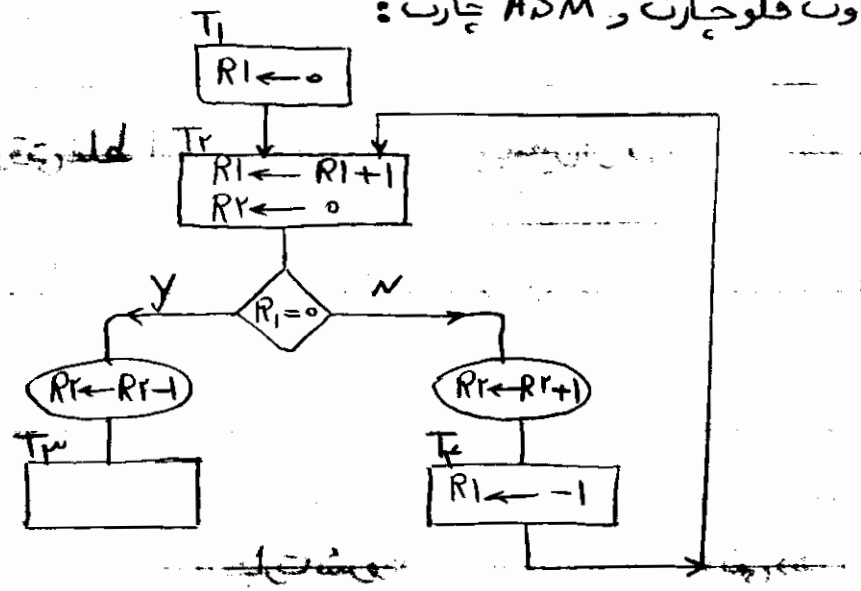
مثال ۲ : آیا ASM چارت زیر درست است ؟



تنها اشکالی که دارد این است که چون μ_2 هم در T_2 و هم در Cen. box آن اجرا می شود ،

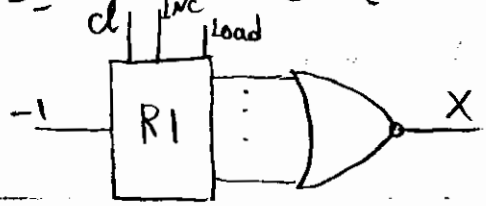
باید μ_2 از state box حالت T_2 حذف شود.

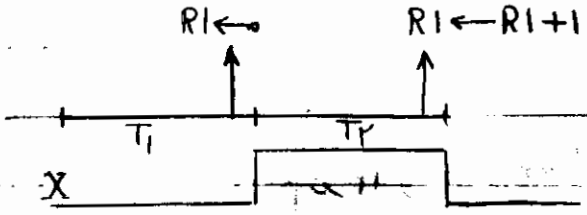
تفاوت فلوچارت و ASM چارت:



اگر شکل بالا را فلوچارت در نظر گرفته و از T_1 شروع کنیم مسیر سمت راست را خواهیم رفت.

امّا اگر ASM چارت در نظر بگیریم مسیر سمت چپ انجام می‌شود. دلیل این است که در لحظه‌ای که کلاک می‌آید (در لبهٔ کلاک) شرط تست می‌شود و در آن لحظه R_1 صفر است. بعد از عبور از لحظهٔ لبهٔ کلاک R1 یک خواهد شد. لذا مسیر سمت چپ انجام می‌شود.

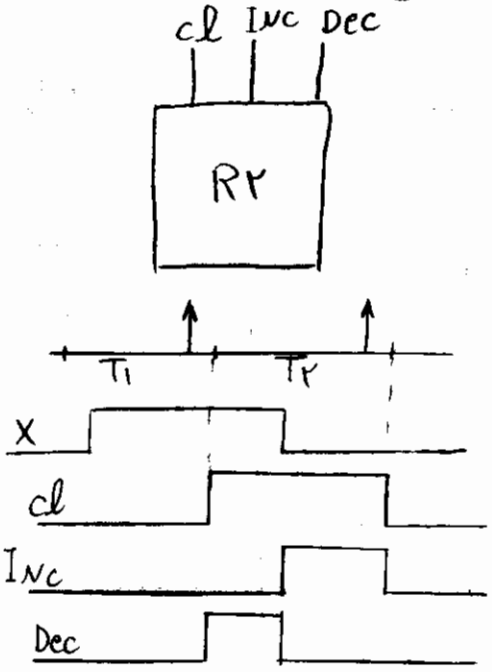




پس می بینیم که مسیر ASM چارت دقیقاً برعکس مسیر فلوچارت است. چون فلوچارت ترتیبی است و ASM چارت همزمان است.

۱- در یک بلوک یک رجیستر می تواند مقدار بگیرد و تست هم بشود.

در فلوچارت اگر مسیر راست را طی کنیم R۲، ۱+ خواهد شد و اگر مسیر سمت چپ را طی کنیم R۲، ۱- خواهد شد. امّا در ASM چارت از هر مسیری که برویم مقدار R۲ نامشخص خواهد بود. حال اگر چنین وضعیتی پیش بیاید:



$cl = T_2$

$Inc = T_2X'$

$Dec = T_2X$

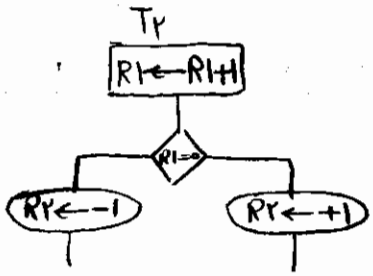
۲- در یک بلوک و در یک مسیر نباید دو OP-M بر روی یک رجیستر داشته باشیم.

حتی اگر در T_2 داشتیم $R2 \leftarrow R2+1$ باز هم غلط است. چون منظور ما ۲ بار اضافه کردن

به R۲ بوده است و نمی توان گفت که به هرحال یکی به R۲ اضافه می شود.

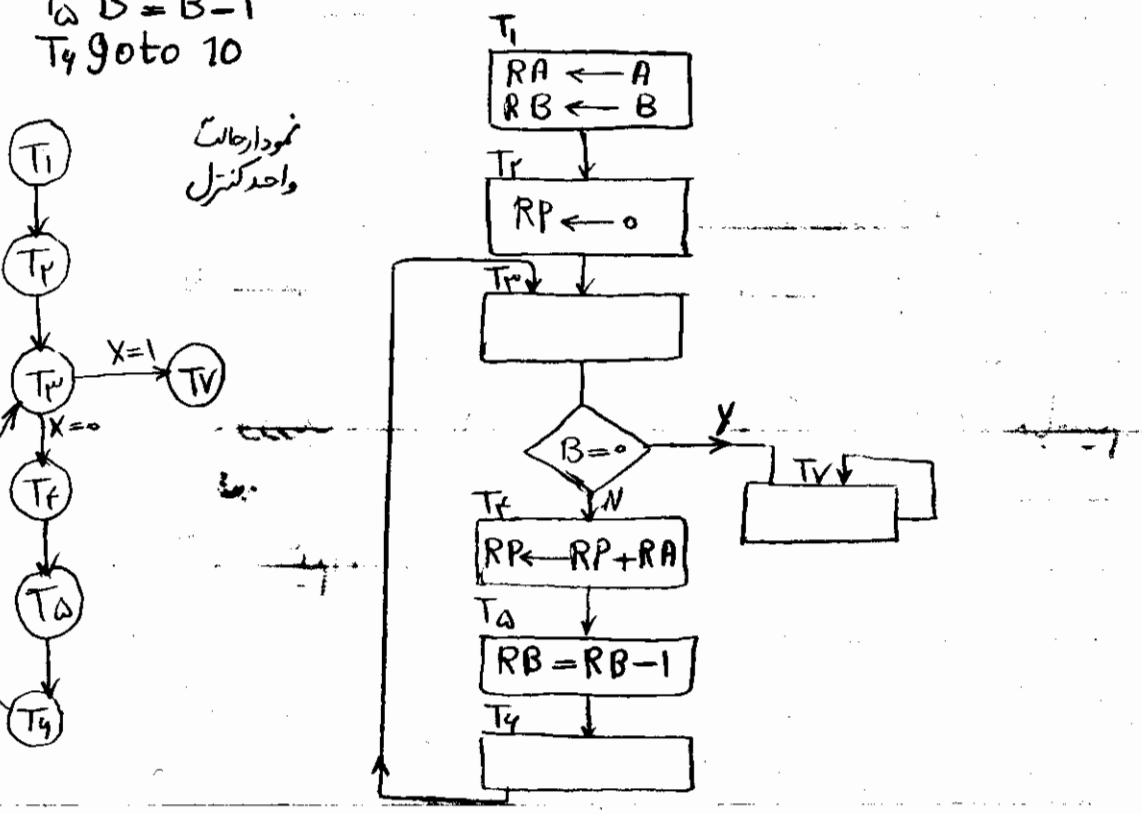
* همچنین در یک بلوک و در یک مسیر برای یک رجیستر نمی توان دو op-μ داشت.

حال اگر چنین نظری داشته باشیم که R۲ در مسیر راست + و در مسیر چپ - و با یک کلاک انجام شود، R۲ ← ۰ را از T۲ حذف کرده و به شکل زیر عمل می کنیم:

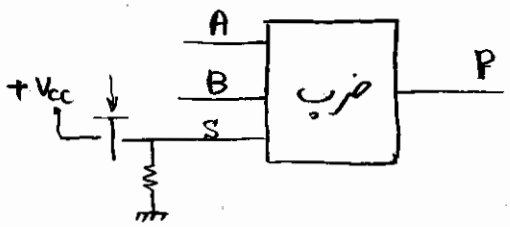


حال مدار ضرب را طراحی می کنیم:

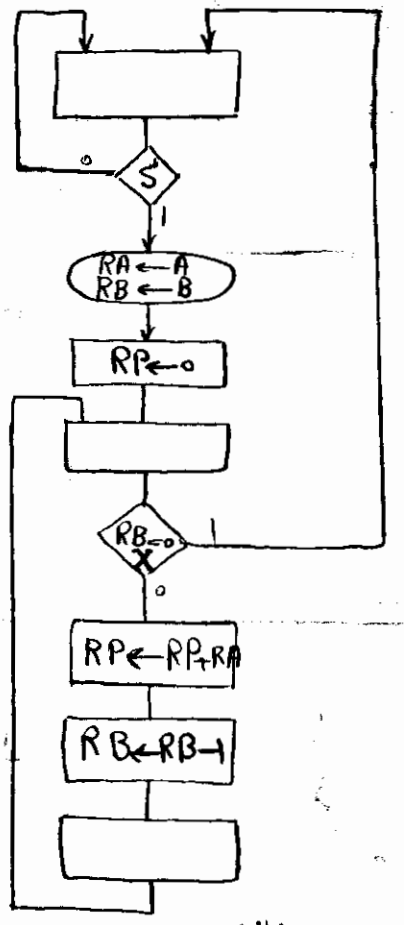
T۱ Read A, B
T۲ P=۰
10 T۳ if B=۰ then stop
T۴ P=P+A
T۵ B=B-۱
T۶ goto 10



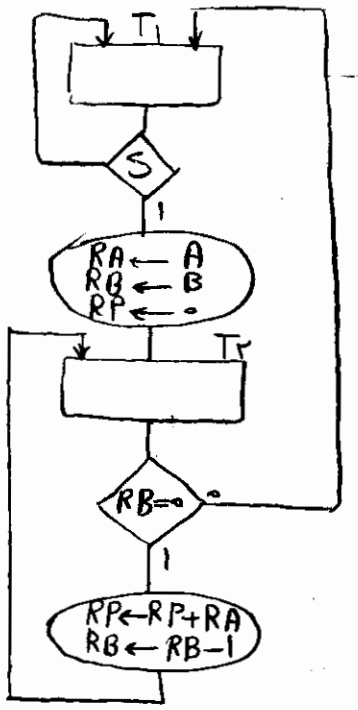
می توان برای توقف مدار، خروجی Y شرط را به T_3 داد. مدار پس از توقف باید خاموش شود تا بتوانیم ضرب دیگری را انجام دهیم. برای اینکه فقط با یک کلید بتوانیم ضرب جدیدی را شروع کنیم، باید به شکل زیر عمل کنیم:




```
20  if s=1 then Read A,B else goto 20
    P=0
10  if B=0 then goto 20
    P=P+A
    B=B-1
    goto 10
```



(الف)



(ب)

$Q_{t+1} =$

ASM چارت (ب) ساده ترین شکل حل مسئله فوق است. نکته ای که باید به آن توجه

داشت این است که در بلوک T_2 RB هم تست می‌شود و هم مقداری گیرد، امّا چون تست با مقدار قبلی RB است، این عمل ایرادی ندارد. اگر μ-op $RB \leftarrow RB-1$ را از Cond. Box برداشته و داخل T_2 خالی بگذاریم جواب ضرب درست است امّا مقدار RB در آخر کار متفاوت با حالت قبل خواهد بود. حال با توجه به ASM چارت

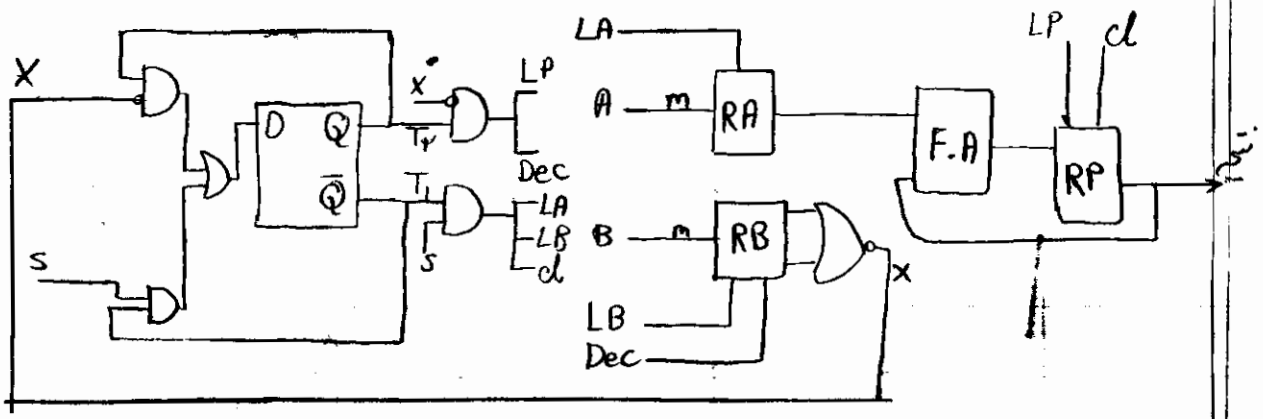
(ب) مدار ضرب را طراحی می‌کنیم:

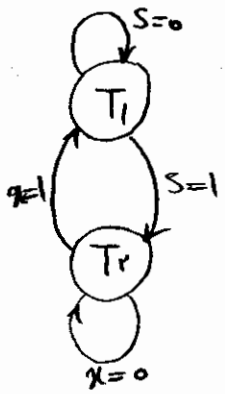
نخست کل μ-op ها را لیست می‌کنیم تا قابلیت‌های هر رجیستر مشخص شود:

$T_1 S$ { $RA \leftarrow A$

$T_2 X'$ { $RB \leftarrow RB-1$
$T_1 S$ { $RB \leftarrow 0$

$T_1 S$ { $RP \leftarrow 0$
$T_2 X'$ { $RP \leftarrow RP+RA$





حالت فعلی Q_t	SX: 00	01	11	10
$T_1 \rightsquigarrow 0$	$T_1 0$	$T_1 0$	$T_2 1$	$T_2 1$
$T_2 \rightsquigarrow 1$	$T_2 1$	$T_1 0$	$T_1 0$	$T_2 1$

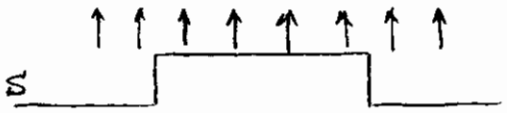
Q_{t+1}

$$D^t = Q^{t+1} = Q'^t . S^t + Q^t X'^t$$

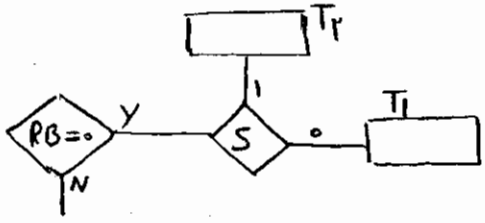
برای D فلیپ فلاپ

با دقت در مدار می توان دید که $\bar{X}$ سنکرون و همزمان با کلاک است یعنی فقط با کلاک تغییر می کند. ولی S ورودی آسنکرون است و اگر فرکانس کلاک کم باشد ممکن است ورودی دیده نشود. از آنجائیکه یک کلاک $\max f_{\text{فرکانس}}$ و $\min f_{\text{فرکانس}}$ دارد، یکی از مواردیکه $\min f$ تعیین می شود پریود سیگنالهای آسنکرون (مثل S) است. $\max f$ راهم که تأخیرات قطعات مدار تعیین می کند.

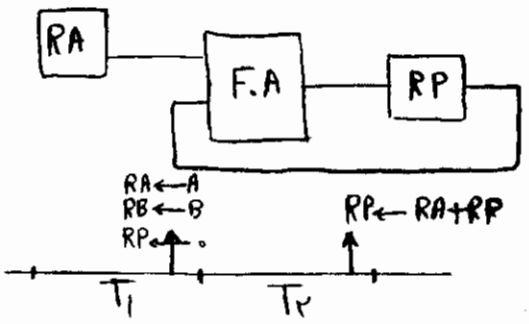
حال اگر فرکانس کلاک را با لا ببریم به شکل زیر خواهد بود:



در این حالت پس از خاتمهٔ عمل باز هم ضرب تکرار می شود تا $S=0$ شود برای اینکه عمل ضرب فقط یکبار انجام شود خروجی Y برای $RB=0$ را به شکل زیر تغییر می دهیم:



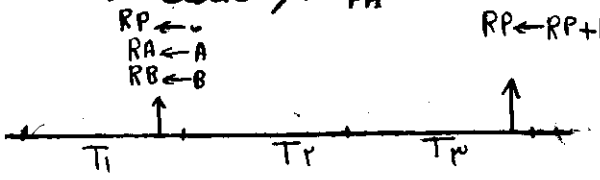
اگر پریود F.A برابر T_{FA} باشد حداقل پریود کلاک چقدر باشد؟



در این حالت باید : $T_{clock} > T_{FA}$

اگر به جای $RP \leftarrow RP+RA$, $RB \leftarrow RB-1$ از T_3: $RP \leftarrow RP+RA$, $RB \leftarrow RB-1$ استفاده کنیم باید :

$T_{clock} > 2T_{FA}$



اگر از یک state box خالی قبل از شرط $RB=0$ استفاده کنیم باید :

$$T_{clock} > 3T_{FA}$$

ولی اگر یک state box بعد از T_3 قرار دهیم باز هم باید :

$$T_{clock} > 2T_{FA}$$

مفید بودن ~~state box~~ های خالی در این جا حس می شود. لذا در طراحی مدار اگر تأخیرات قطعات متفاوت باشند لازم نیست حتماً با کندترین خودمان را تطبیق دهیم بلکه می توانیم از state box های خالی در مدار استفاده کنیم و پریود مناسب را برای کلاک انتخاب کنیم.

روشهای ساخت واحد کنترل :

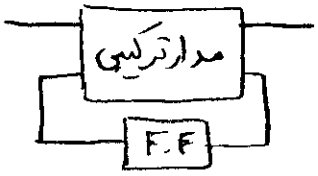
۱- کلاسیک ← مزیت: آشنا ← عیب: IC متعدد و بی نظمی

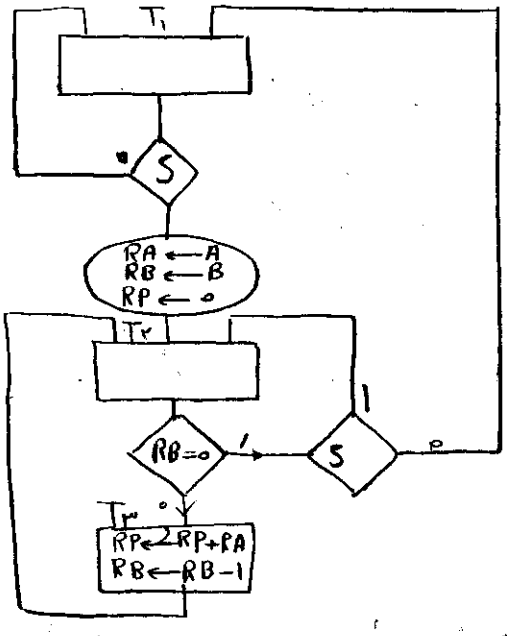
۲- F.F ۱ برحالت ← مزیت: سادگی ← عیب: IC متعدد و بی نظمی

۳- Mux - Reg - dec
قطعات زیاد ولی منظم

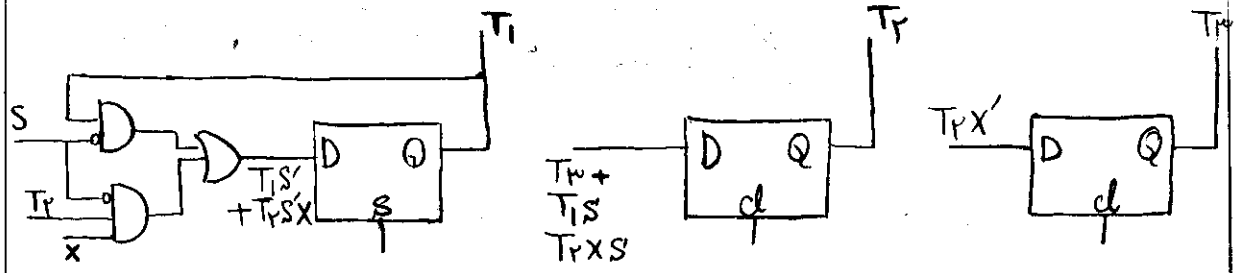
۴- PLA - Reg
حتمی
استفاده از ۲ Ic

۵- Rom - Reg
استفاده از دو Ic





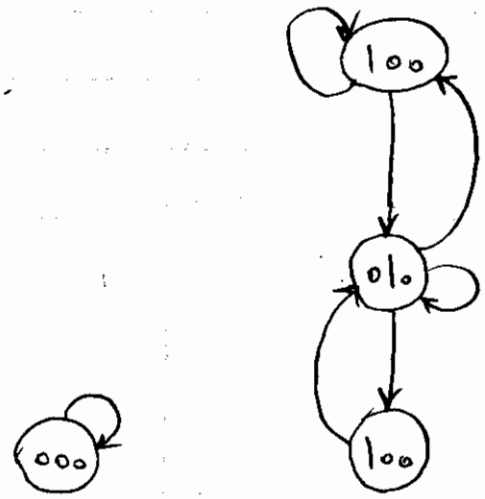
روش FF ۱ برحالت :



اما ورودی هامی توانند ساده‌ترهم بشوند. این روش حتماً تضمین می کند که هیچگاه T_1 و T_2 و T_3 باهم یک نمی‌شود و هر وقت مثلاً T_1 یک باشد بقیه صفرهستند.

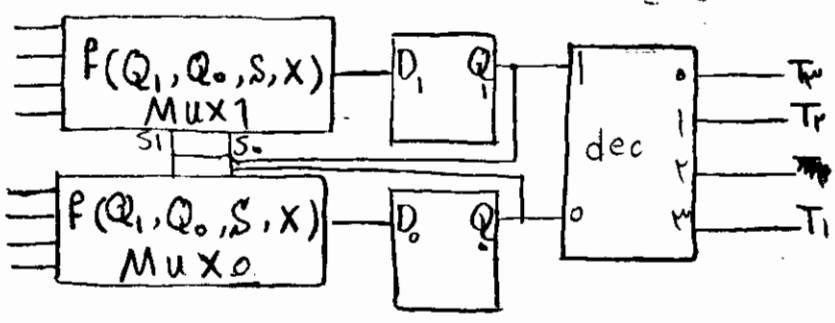
ایرادی که این مدار دارد این است که حالتهای استفاده نشده داریم یعنی اگر

به یک حالت استفاده نشده برویم تکلیف چه خواهد بود. در مدار فوق ۸ حالت داریم که فقط ۳ حالت آنها استفاده شده است.



در مدار فوق به عنوان مثال اگر به حالت ۰۰۰ برویم یعنی T_1 و T_2 و T_3 صفر بشند هیچ‌گاه از آن خارج نخواهیم شد. لذا باید حالتهای استفاده نشده را آنالیز کنیم. در این روش طراحی واحد کنترل اصلاح حالتهای استفاده نشده سخت و پیچیده خواهد که از معایب آن محسوب می‌شود. همچنین اگر به علت نویز به یک حالت استفاده شده دیگری که نمی‌خواهیم، برویم دیگر قابل تشخیص نیست.

روش Mux-Reg-Dec : مثلاً if $T_1=11$, $T_2=01$, $T_3=00$



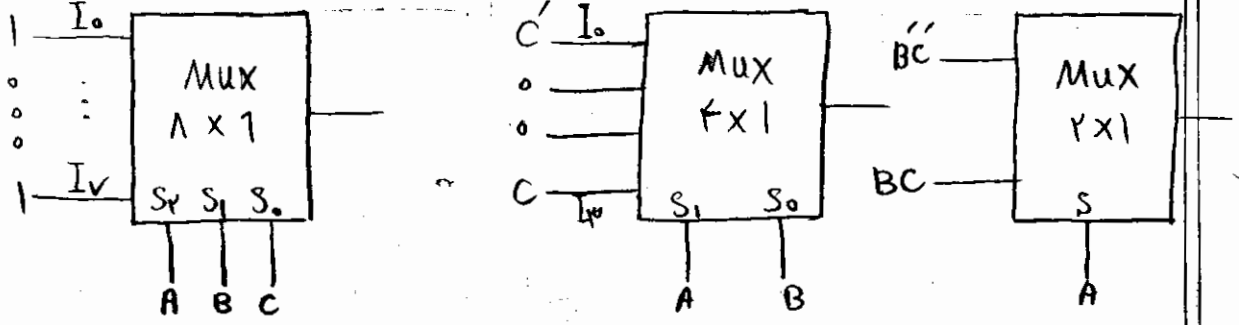
Mux – Reg – Dec

برای طرح مدار ترکیبی n متغیره از Mux حداقل با یک خط select تا n خط

می توانیم استفاده کنیم :

مثال :

$F = A'B'C' + ABC$



* در این روش به تعداد فلیپ فلاپ ها خط select برای MUX قرار می دهیم.

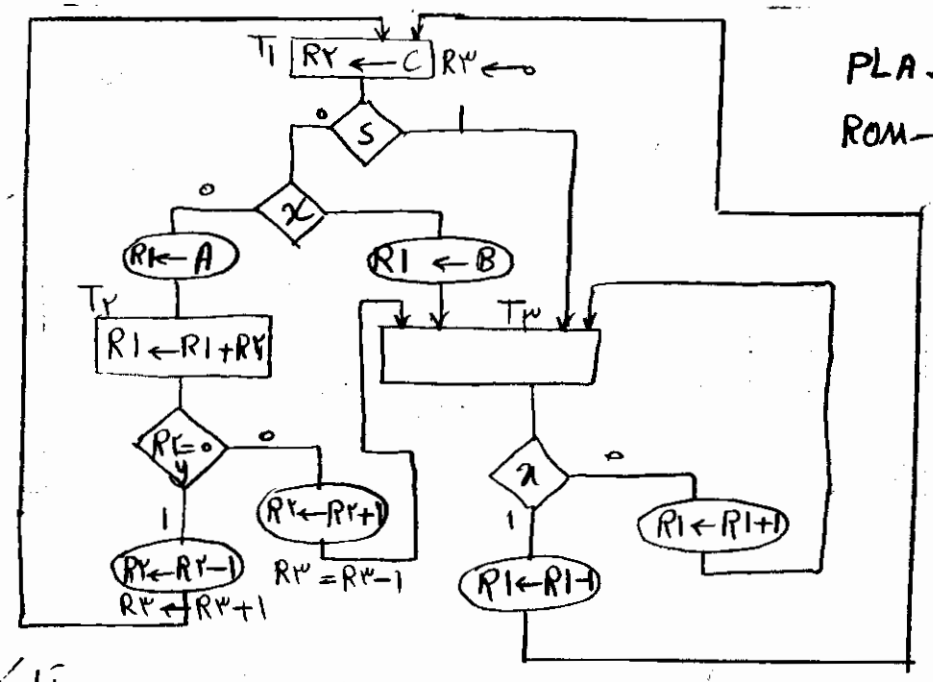
برای بدست آوردن ورودی های MUX برای مثال فوق به شکل زیر عمل می کنیم :

حالت فعلی Q_1^t	Q_0^t	حالت بعدی Q_1^{t+1}	Q_0^{t+1}	شرط تابع بولی	ورودی MUX۱		ورودی MUX۲	
۱	۱	۱	۱	S'	I_3	S'	I_3	۱
۱	۱	۰	۱	S				
۰	۱	۰	۰	X'	I_1		I_1	
۰	۱	۰	۱	XS		XS'		$XS + XS' = X$
۰	۱	۱	۱	XS'				
۰	۰	۰	۱	۱	I_0	۰	I_0	۱

شرط ها در هر بخش :

۱- AND دو به دوی آنها صفر است. ۲- OR دو به دوی آنها یک است

درغیراینصورت ASM چارت ناقص بوده یا مشکل دارد.



PLA_Reg -۴

ROM-Reg -۵

توابع کنترلی

$T_1 : RY \leftarrow C$

$T_2 y' : RY \leftarrow RY + 1$

$T_2 y : RY \leftarrow RY - 1$

$T_1 S' x' : RI \leftarrow A$

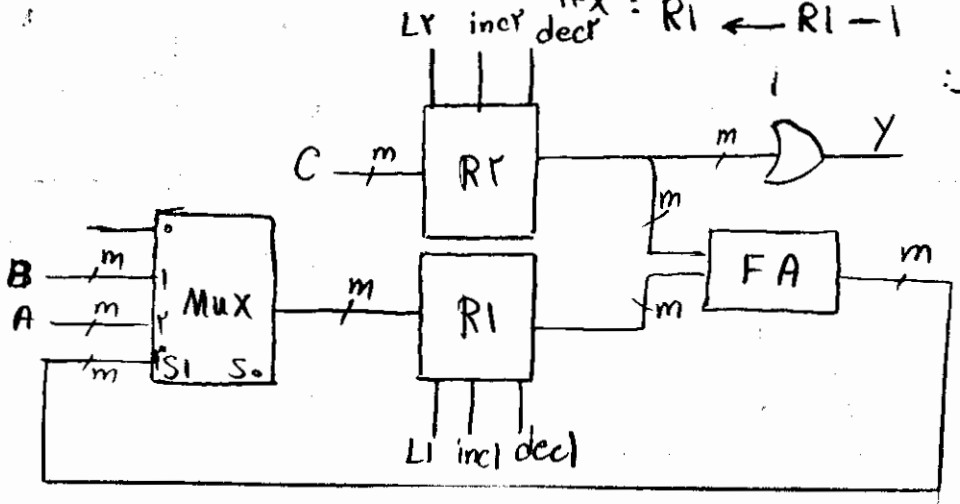
$T_1 S' x : RI \leftarrow B$

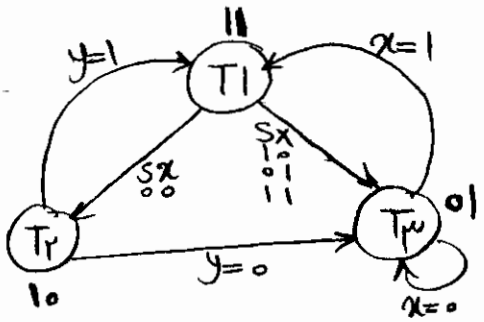
$T_2 : RI \leftarrow RI + RY$

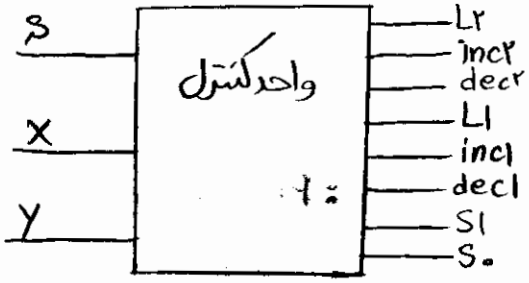
$T_3 x' : RI \leftarrow RI + 1$

$T_3 x : RI \leftarrow RI - 1$

واحد پردازشگر:







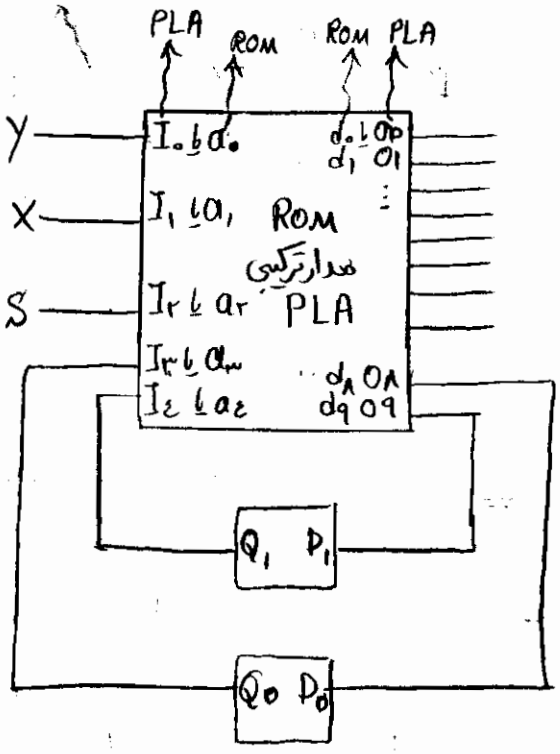
$= T_1$
$T_2 y'$
$T_2 y$
$T_2 + T_1 S'$ → $T_1 S' x' + T_1 S' x$
$T_3 x'$
$T_3 x$
$* T_1 S' x' + T_2$
$* T_1 S' x + T_2$

اگر جلی مسیر ۳ و ۰ را در Mux عوض کنیم آن گاه :

برای ساده شدن این حالت را در نظر می گیریم:

$S_1 = T_1 S' x'$ ، $S_0 = T_1 S' x$

در حال حاضر واحد کنترل ۸ خروجی دارد . اگر تعداد op-μ ها زیاد شود باز هم خروجی ها عوض نمی شود. (op-μ های جدید با رنگ قرمز مشخص شده اند) یعنی هیچ لزومی ندارد تعداد خروجی ها را زیاد کنیم . در طراحی همیشه تعداد خروجی های متفاوت را در نظر می گیریم .



PLA- Reg و ROM - Reg

حُسن این روشها این است که کل مدار ترکیبی در یک IC جای می گیرد.

جدول حالت خلاصه شده برای PLA :

	حالت فعلی Q_1^t	حالت فعلی Q_0^t	ورودی ها S	ورودی ها X	ورودی ها Y	حالت بعدی Q_1^{t+1}	حالت بعدی Q_0^{t+1}	خروجی ها L_2	incr	decr	L_1	incl	del	SI	S_0
	I_4	I_3	I_2	I_1	I_0	O_9	O_8	O_7	6	5	4	3	2	1	Q_0
T_1	1	1	0	0	—	(1)	1	1	0	0	1	0	0	1	0
	1	1	0	1	—	0	1	1	0	0	1	0	0	0	1
	1	1	1	—	—	0	1	1	0	0	0	0	0	0	0
T_2	1	0	—	—	0	0	1	0	1	0	1	0	0	0	0
	1	0	—	—	1	(1)	1	0	0	1	1	0	0	0	0
T_3	0	1	—	0	—	0	1	0	0	0	0	1	0	0	0
	0	1	—	1	—	(1)	1	0	0	0	0	0	1	0	0
						D_0^t	D_1^t	T_1		$T_2 y$	$T_2 + T_1 S'$				

اگر از PLA استفاده نکنیم ورودی های فلیپ فلاپ ها به صورت زیر خواهد بود:

$$D_1 = Q_1 Q_0 S' X' + Q_1 Q_0' y + Q_1' Q_0 X$$

$$D_2 = (Q_1 Q_0 S' X')'$$

معمولاً در جدول برنامه ریزی PLA صفرهای خروجی را با — نمایش می دهند.

تعداد سطرهای جدول PLA برابر تعداد مسیرهای وارد شده به state box ها است.

جدول فوق نیاز به یک PLA با 7 گیت AND و 4 گیت OR است.

تعداد ستونهای خروجی — تعداد سطرها

می‌کنیم.

اگر بخواهیم از سیر مکمل (F) خارج شویم جای صفرها و یک ها را در آن ستون عوض

در Mux دیدیم که آنالیز و تصحیح حالتهای استفاده نشده بسیار راحت بود.

در مدار فوق ۲ فلیپ فلاپ داریم و ۴ حالت که یک حالت آن یعنی ۰۰ استفاده نشده

است. از روی جدول می‌توان گفت حالت بعد از حالت استفاده نشده چیست. در

مدار فوق اگر مدار به حالت ۰۰ برود، تمام گیت‌های AND ، خروجی صفر می‌گیرند

ولذا مدار در همان حالت ۰۰ باقی می‌ماند.

در حالت کلی به ازای هر حالت استفاده نشده تمام گیت‌های AND صفر می‌شوند و در

نتیجه تمام خروجی‌های PLA مقدار صفر می‌گیرند.

حال برای اینکه از حالت استفاده نشده خارج شویم دو راه حل داریم:

۱- اگر مثلاً می‌خواهیم به T_2 برویم، T_2 را ۰۰ در نظر می‌گیریم.

۲- یک سطر به شکل زیر به جدول اضافه کنیم:

T_4	۰۰	— — —	حالتی که می‌خواهیم	...

در این حالت به هر حال یک گیت AND اضافه می‌شود.

حال اگر بخواهیم بدون اضافه کردن AND مشکل را حل کنیم به شکل زیر عمل می‌کنیم:

O_9 O_8

TC

فرض کنیم اگر به ۰۰ رفتیم می‌خواهیم به ۱۰ برود. در این صورت ستون دوم خروجی را همانطور که به رنگ قرمز نشان داده شده تغییر می‌دهیم.

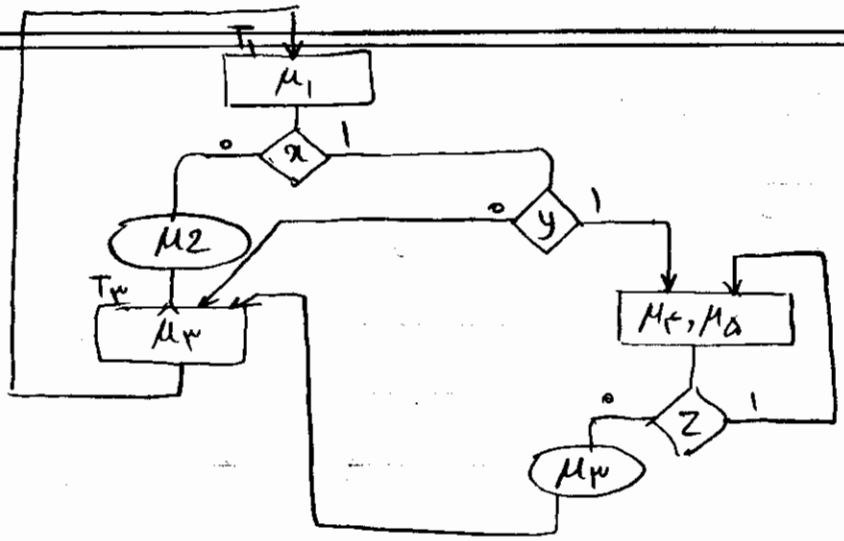
وقتی که یک ستون را Complement می‌کنیم در نقاط تعریف شده (حالات استفاده شده) تابع تغییری نمی‌کند، ولی در نقاط تعریف نشده ستونهای C دارای مقدار یک می‌شوند.

در طراحی با ROM جدول به شکل زیر تغییر می‌کند:

	Q_1	Q_0	S	x	y									
	a_4	a_3	a_2	a_1	a_0	d_9	d_8	...						d_0
0 →	0	0	0	0	0									
1 →	0	0	0	0	1									
⋮				حالت استفاده نشده										
7 →														
⋮														
17 →	1	0	0	0	1	1	1	d_5		...				
⋮														
31 →	1	1	1	1	1	0	1							

برای تعیین d_9 تا d_0 از روی ASM چارت می‌توانیم دنبال کنیم و حالت بعدی را پیدا کنیم و همچنین با تشخیص μ-op هایی که انجام می‌شوند d_7 تا d_0 را تعیین کنیم. ۲- از روی جدول PLA مقادیر را بیابیم.

مثال :

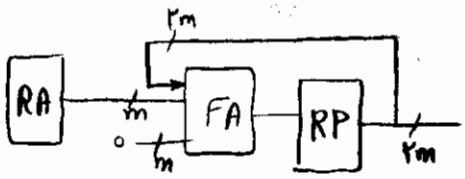


برای طراحی مدار فوق با PLA به یک PLA دارای ۵ ورودی (تعداد state box ها + تعداد Condition box) و دارای ۶ خروجی (تعداد خرمان های متفاوت + تعداد خروجی فلیپ فلاپها) و نیاز به ۶ گیت AND (تعداد مسیرهای وارده به state box) دارد.

در طراحی با ROM ، ۲۴ کلمه تعیین تکلیف می شوند که ۸ تای آنها استفاده نشده است.
مبهم (سوال امتحانی)

مثال :

```
Read A, B
P = 0
10 if B=0 then stop
   P = P + A
   B = B-1
   goto 10
```



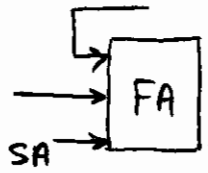
آیا الگوریتم بالا اعداد منفی را هم ضرب می کند؟

برای این منظور باید قرارداد دیگری برای اعداد منفی را بدانیم که عموماً مکمل ۲ است. در این صورت باید تغییرات مقابل را بدهیم :

if B>0 then $\begin{matrix}P=P+A\\B=B-1\end{matrix}$ else $\begin{matrix}P=P-A\\B=B+1\end{matrix}$

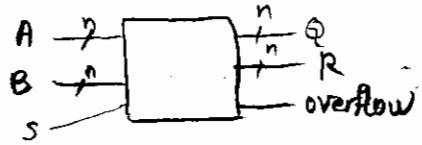
اگر B مثبت باشد الگوریتم را تغییر نمی‌دهیم. فقط صفرهای ورودی به FA اگر A>0
بیت علامت
باشد باید صفر باشند و اگر A<0 باید ۱ باشند. بیت اول عدد که بیت علامت است

با S نمایش داده می‌شود:

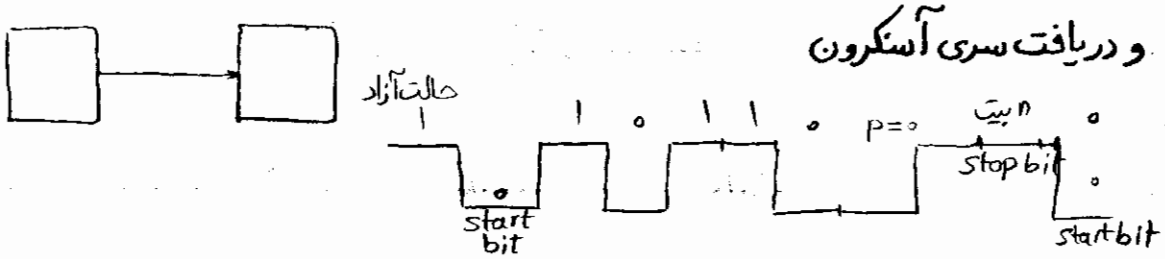


تکلیف ۱: فصل ۸ مانو (۴و۱۲، ۲۰، ۲۱، ۲۲) و مدار ضرب با علامت (با قرارداد

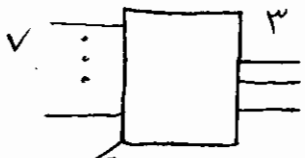
مکمل ۲) و تقسیم با علامت.



و دریافت سری آسنکرون

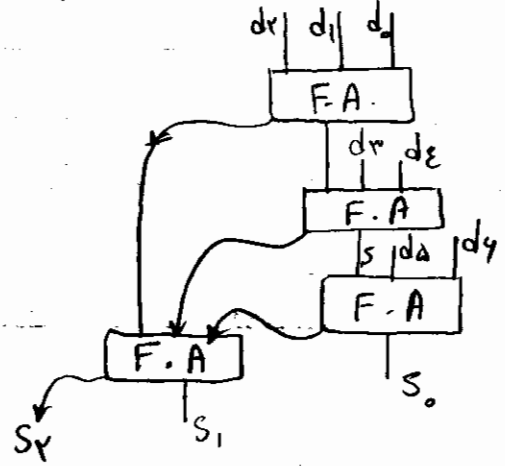


مثال: خروجی سیستم زیر تعداد ۱های ورودی را نشان می‌دهد.

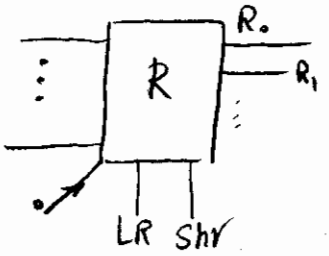


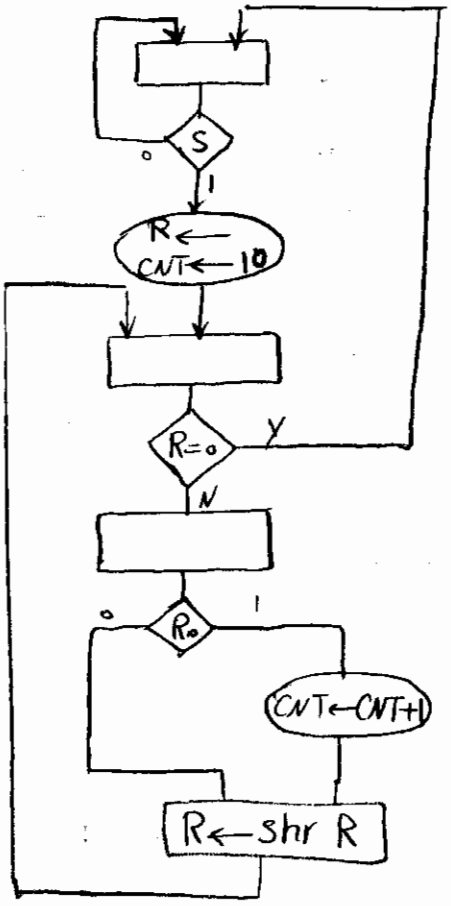
آیا می‌توان مدار فوق را ترکیبی طراحی کرد؟ بلی. به این ترتیب که تمام ورودی‌ها را با

هم جمع می‌کنیم.



طراحی ترتیبی مدار فوق : اگر بخواهیم تک تک ورودی ها را چک کنیم به ازای n ورودی نیاز به n شرط در ASM چارت پیدا می کنیم. لذا از یک شیفت رجیستر استفاده می کنیم وفقط یک خروجی را تست کرده وبعد از هر بار یکبار شیفت می دهیم. شرط خاتمه را صفر می گذاریم و به اطلاعات هم نیازی نداریم (بعد از شمارش).

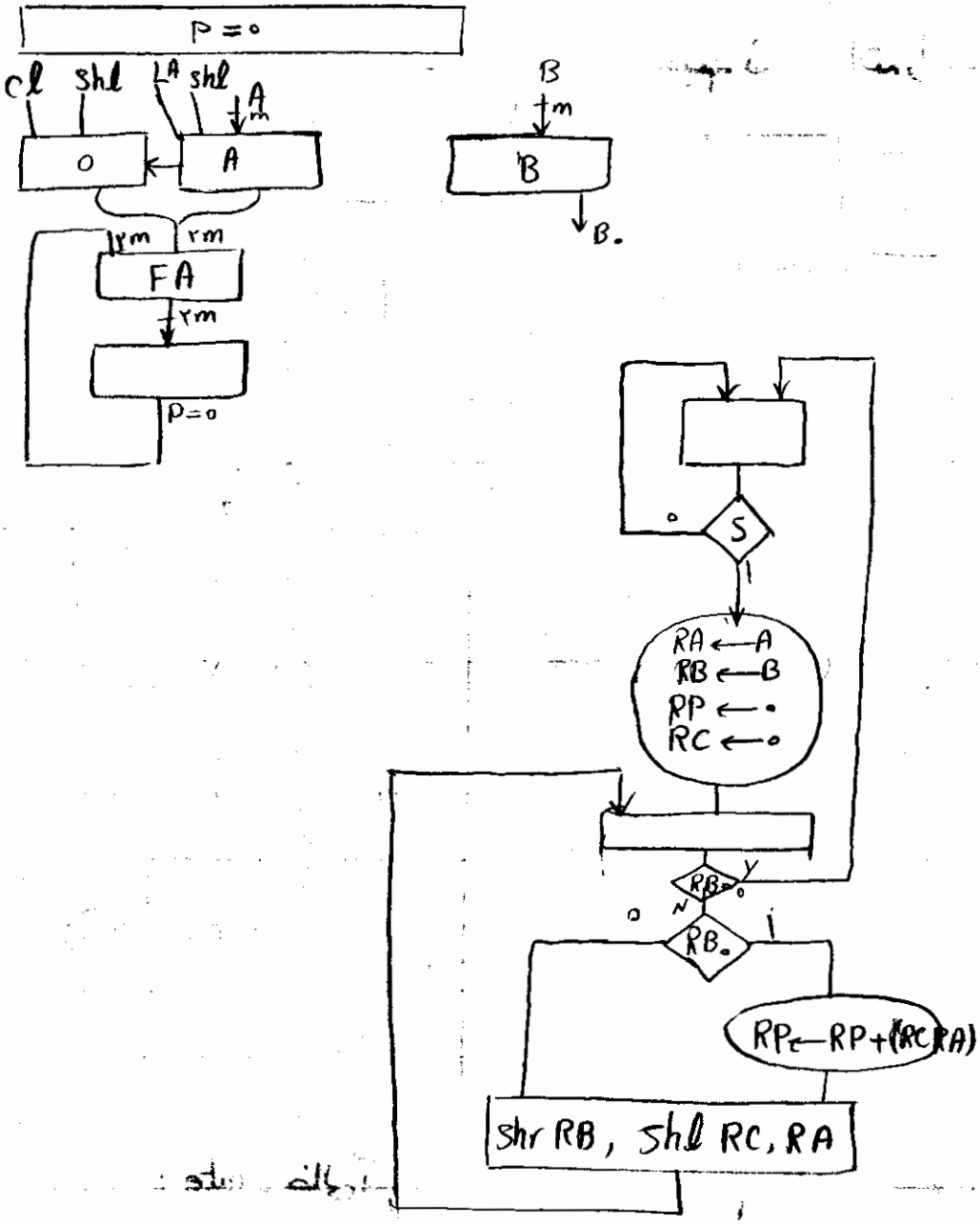




برای ساده سازی state box خالی آخر را حذف می کنیم. می توانیم R ← Shr R را از Condition box برداشته و در state box خالی دوم قرار دهیم و تنها فرقی که دارد در پایان، هنگام خروج یکبار هم به سمت راست شیفت خواهیم داشت.

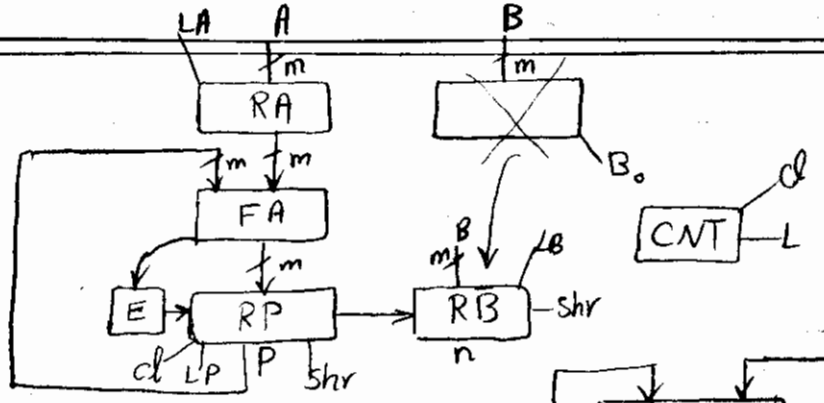
مثالی دیگر برای ضرب:

$$\begin{array}{cccccc} & & & a_2 & a_1 & a_0 \\ & & & b_2 & b_1 & b_0 \\ \hline 0 & 0 & 0 & a_2 & a_1 & a_0 & b_0 \\ 0 & 0 & a_2 & a_1 & a_0 & 0 & b_1 \\ 0 & a_2 & a_1 & a_0 & 0 & 0 & b_2 \end{array}$$

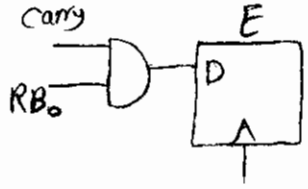


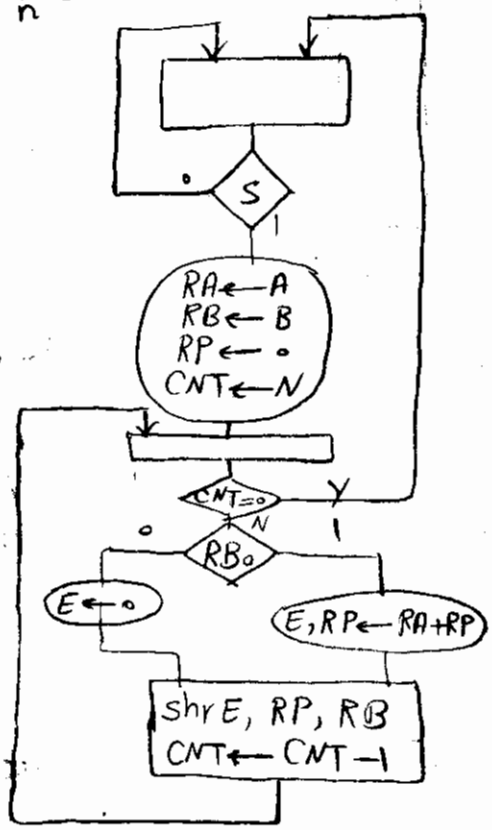
برای ساده سازی می توانیم state box آخر را با Condition box یکنیم.

روش دیگر طراحی:



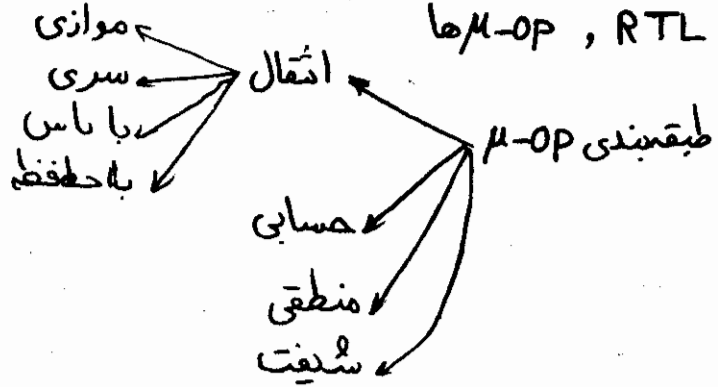
در اینجا مدار ساده نمی‌شود.



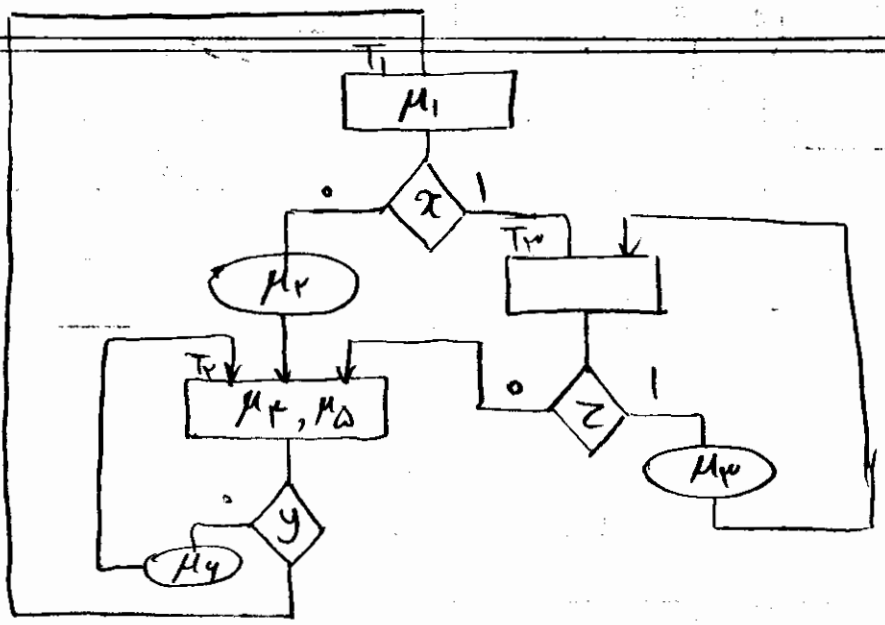


فصل ۴ :

RTL , μ-OPها



کنترل



فرمان : ها μ-op

$$* \begin{cases} T_1 x & : \ \text{goto } T_3 \\ T_1 & : \ \mu_1 \ , \\ T_1 x' & : \ \mu_2 \ , \ \text{goto } T_2 \end{cases}$$

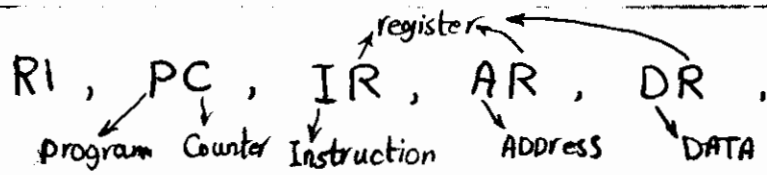
T_2 : μ_4, μ_5

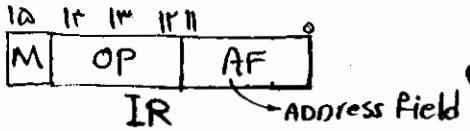
$T_6 y'$: μ_6

$T_3 z$: μ_3

$*$ T_1 : μ_1 , if (x) then (goto T_3) else {goto T_2, μ_2}

قراردادهای زبان :

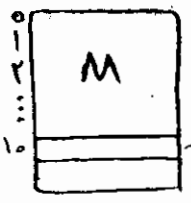




نحوهٔ شماره گذاری بیت های رجیستر

IR_{15} تعیین بیت خاص رجیستر

$IR(0-11) = IR(AF)$, $IR(12-14) = IR(OP)$



درمورد حافظه :

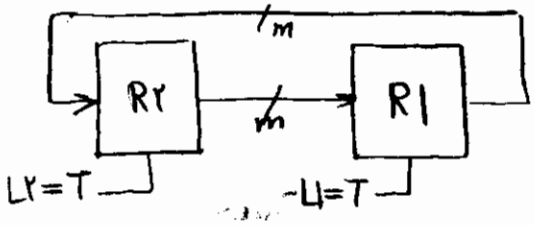
کلمه دهم حافظه $M[10]$

μ-OP‌ها : فرمان

T :

μ_4 ، μ_5 : تابع بولی

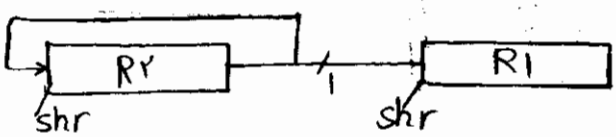
μ-OP‌های انتقال موازی :



$T : R1 \leftarrow R2 \ , \ R2 \leftarrow R1$

شرط انجام فرامین بالا این است که رجیسترها حتماً حساس به لبه باشند.
(edge triggered)

انتقال سری :



با توجه به اینکه در انتقال مبدأ نباید تغییر کند لذا مدار R۲ دارای فیدبک است.

در انتقال سری برای n بیت نیاز به n کلاک داریم،

$$T: \quad R2_i \leftarrow R2_{i+1} \ , \ R1_i \leftarrow R1_{i+1} \ , \ i=0,\dots,n-2$$
$$R2_{n-1} \leftarrow R2_0 \ , \ R1_{n-1} \leftarrow R2_0$$

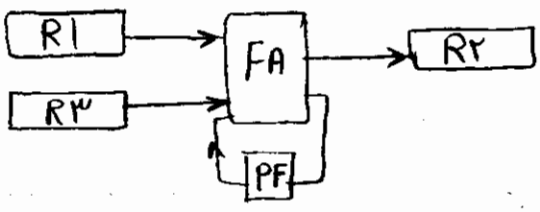
word time

WT_1: $R1 \leftarrow R2$, انتقال سری

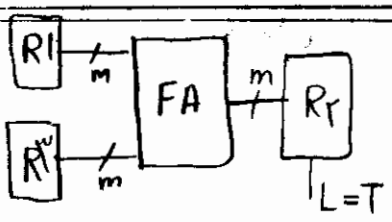
اعلان

اگر « انتقال سری » را نگذاریم به معنی انتقال موازی خواهد بود.

WT_2 : $R2 \leftarrow R1 + R3$, انتقال سری



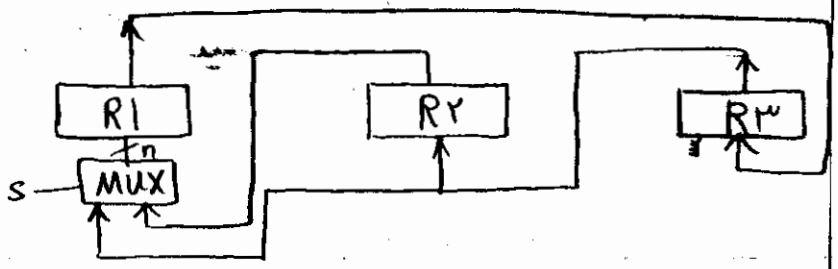
T: $R2 \leftarrow R1 + R3$



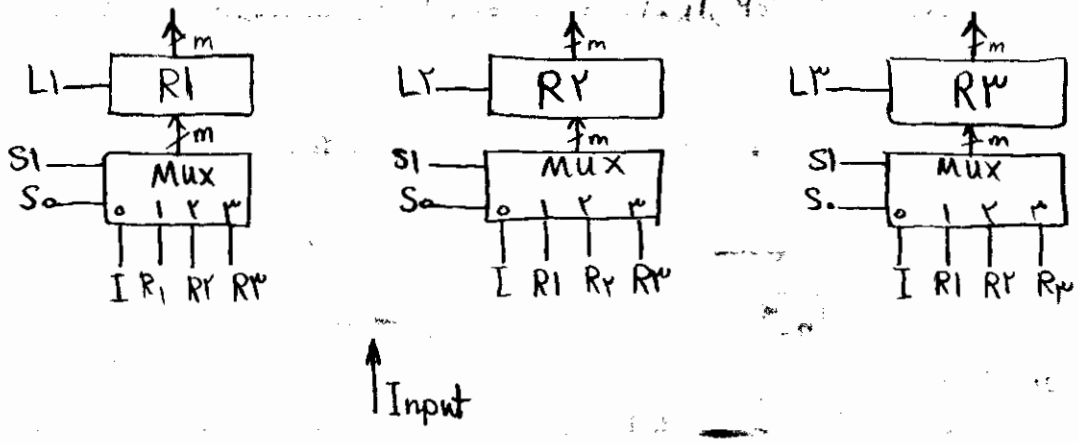
انتقال با باس:

هدف امکان انتقال از هر رجیستر به هر رجیستر.

$R1 \leftarrow R2$
$R2 \leftarrow R3$
$R3 \leftarrow R1$
$R1 \leftarrow R3$



در حالت کلی:

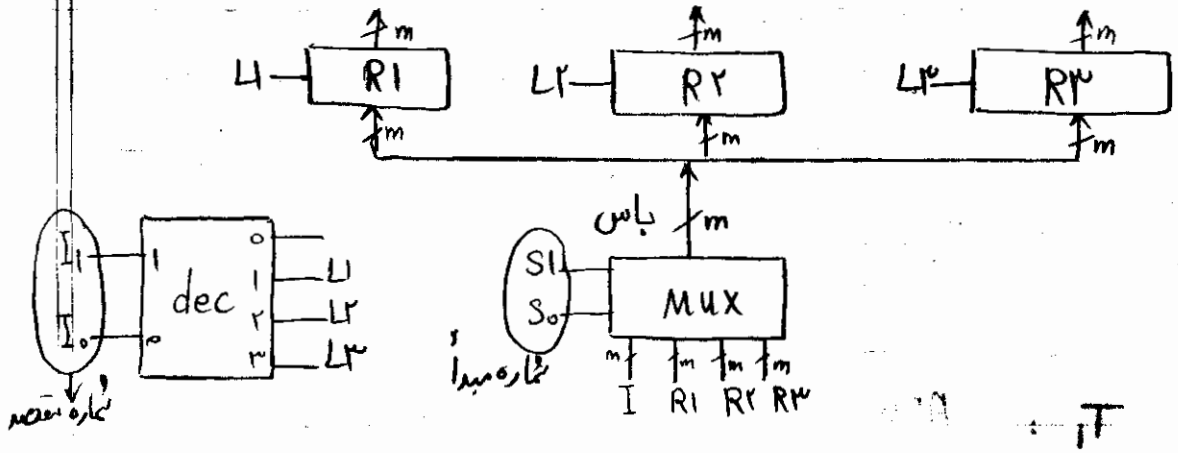


Input

در این طرح اگر تعداد رجیسترها M باشد و تعداد بیت هر رجیستر N باشد آن گاه، $M \times N$ تعداد Mux های استفاده شده است و تعداد خطوط Data برابر $M^2 \times N$ است. تعداد خطوط فرمان واحد کنترل برابر $M + KM$ است. M انتقال همزمان و مستقل می توانیم داشته باشیم. برای اینکه در هر کلاک M انتقال نداشته باشیم

مفهوم باس مطرح می‌شود. در باس ایدهٔ مسیر عمومی بیان می‌گردد. درحالیکه در

فرم بالا برای هر رجیستر یک مسیر خصوصی داریم.



در این حالت تعداد MUX ها به تعداد بیت ها یعنی N است. تعداد خطوط DATA برابر

$N \times M$ ، تعداد فرمان‌ها $K+M$ یا $K+K$ است.

یک مبدا و مقصد متعدد — یک مبدا و یک مقصد

درحالتی که فقط یک مقصد داریم از یک decoder به فرم بالا استفاده می‌کنیم.

با تک مقصد امکان پذیر نیست ← $T_1 : R_2 \leftarrow R_1 \ , \ R_3 \leftarrow R_1$

چون دو مبدا داریم. غ $T_2 : R_1 \leftarrow R_3 \ , \ R_3 \not\leftarrow R_2$

باز هم دو مبدا داریم. غ $T_3 : R_1 \leftarrow R_2 \ , \ R_2 \not\leftarrow R_1$

برای انجام T_2 نیاز به دو کلاک داریم، و برای انجام T_3 به یک رجیستر اضافی (کمکی)

و سه کلاک نیاز داریم.

در لیست op-m ها اگر اعلانی نباشد به مفهوم مسیر خصوصی است. امّا می توان با یک

اعلان مانند ABUS نحوهٔ انجام را از نوع مسیر عمومی کرد.

در مثالی که زده شد:

وقتی در T_1 هستیم

مبدا
$S_1 = T_1 + T_2 + 0 + T_1$
$S_2 = 0 + T_2 + T_3 + 0$

مقصد
$I_1 = 0 + T_2 + T_3 + T_4$
$I_2 = T_1 + 0 + T_3 + T_4$

وقتی در T_2 هستیم.

T_1 : ABUS ← R2

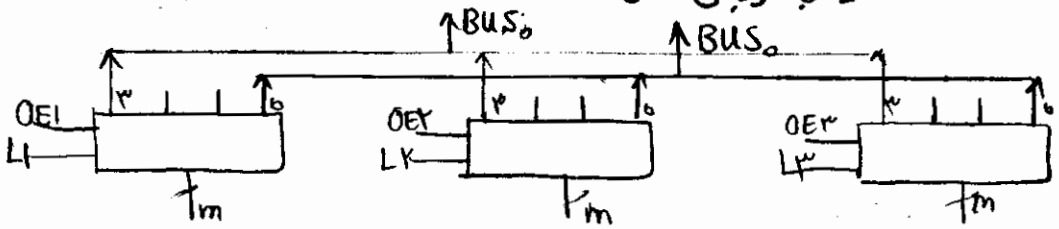
T_2 : R3 ← ABUS

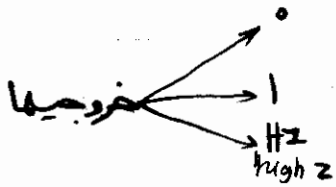
غ حون BUS حافظه ندارد

T_1 : R3 ← R2 , >
انتقال از طریق باس

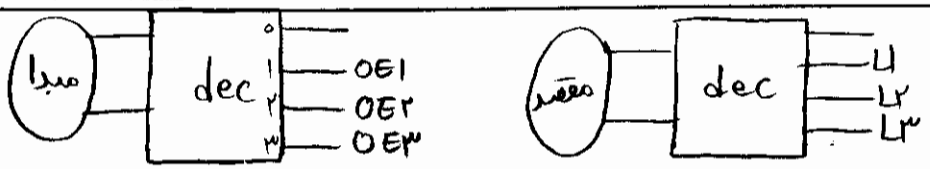
BUS حافظه ندارد.

تشکیل باس با MUX ← با خروجی های سه حالته.





مدارهایی که سه حالته هستند معمولاً یک OE (output enable) دارند، که باعث فعال شدن می شود.



دراین روش بجون نیازبه MUX ، BUS راتشکیل می‌دهیم.

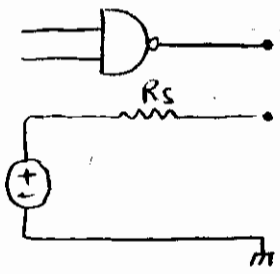
در مدارهای زیر اتصال خروجی‌ها مجاز نیست:

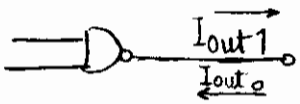
خروجی سه‌حالته
3-state ___ تشکیل باس

ECL ___ wired OR

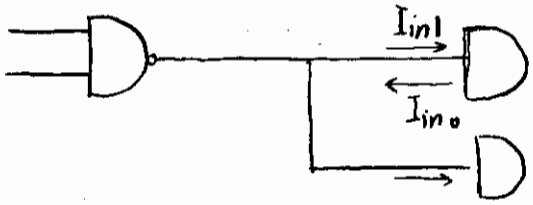
TTL . open collector ___ wired AND

تشکیل باس { با MUX / با خروجی سه‌حالته.





I_{out1} حدجریانی است که می‌توانیم ازگیت بکشیم تا آن عوض نشود. I_{out0} حدجریانی است که گیت می‌تواند بکشد تا آن عوض نشود.



$$F_{out_1} = \frac{I_{out_1}}{I_{in\,1}} \quad , \quad F_{out_0} = \frac{I_{out_0}}{I_{in\,0}}$$

Fout نشان می‌دهد که یک خروجی چند ورودی را حداکثر می‌تواند تأمین کند.

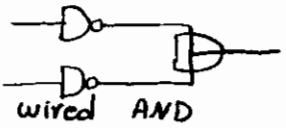
خروجی سه‌حالته

0 Lz
1 Lz
Hz

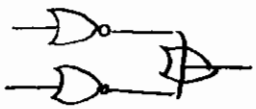
در حالت Hz مدار دیگر نمی‌تواند جریان لازم برای بار را تأمین کند.

همان طور که گفتیم در حالتهای زیر اتصال خروجیها مجاز نیست:

TTL open-collector

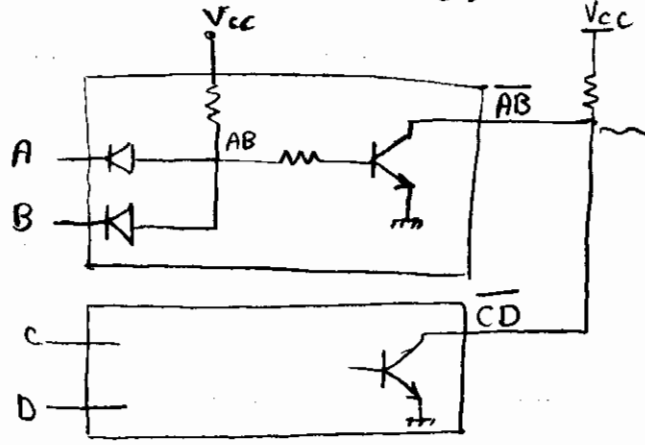


ECL



خروجی سه‌حالته — تشکیل باس

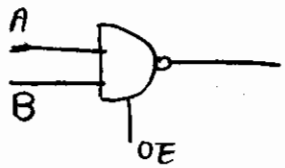
در خروجی سه‌حالته، اتصال خروجیها باعث تشکیل باس خواهد شد.

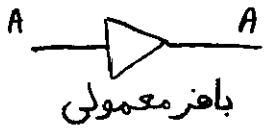


مدلی از open-collector TTL :

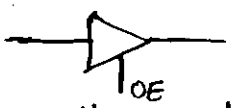
ورودی OE (output enable) در IC ها برای خروجی Hz است. هنگامیکه OE صفر است خروجی Hz و هنگامی که OE یک باشد خروجی فعال و Lz است.

لذا گیت‌هایی که دارای خروجی سه‌حالته هستند بشکل زیر هستند:



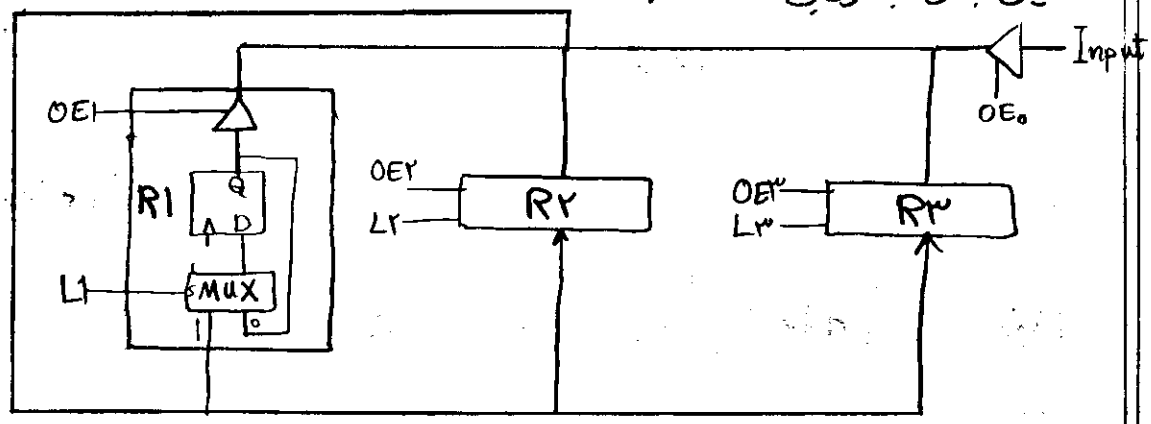


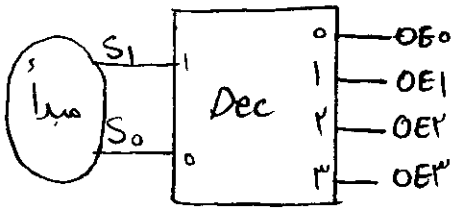
بافر معمولی

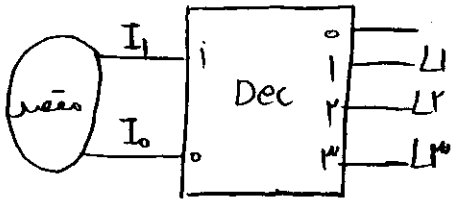


بافر سه حالته

تشکیل باس با خروجی سه حالته :







دراین روش به جای n تا Mux ۴x۱ از یک Dec ۲x۴ استفاده می‌شود.

T_0: $R1 \leftarrow R2$, $R1 \leftarrow Bus$, $Bus \leftarrow R2$
T_1: $R3 \leftarrow R1$

درست.

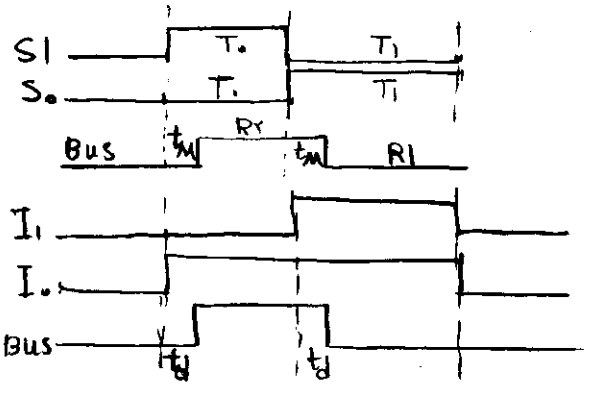
T_0: $R1 \leftarrow R2$, $Bus \leftarrow R2$
T_1: $R3 \leftarrow R1$ $R1 \leftarrow Bus$

غلط.

T_0: $R1 \leftarrow R2$ $Bus \leftarrow R2$
T_1: $R3 \leftarrow R1$ $R1 \leftarrow Bus$, $Bus \leftarrow R2$

درست.

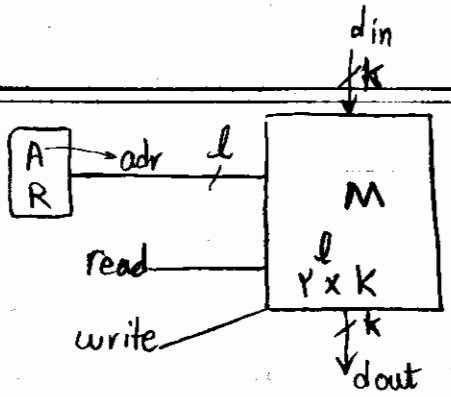
$S1 : T_0 + 0$
$S_0 : 0 + T_1$
$I_1 : 0 + T_1$
$I_0 : T_0 + T_1$



t_M : تأخیر Mux

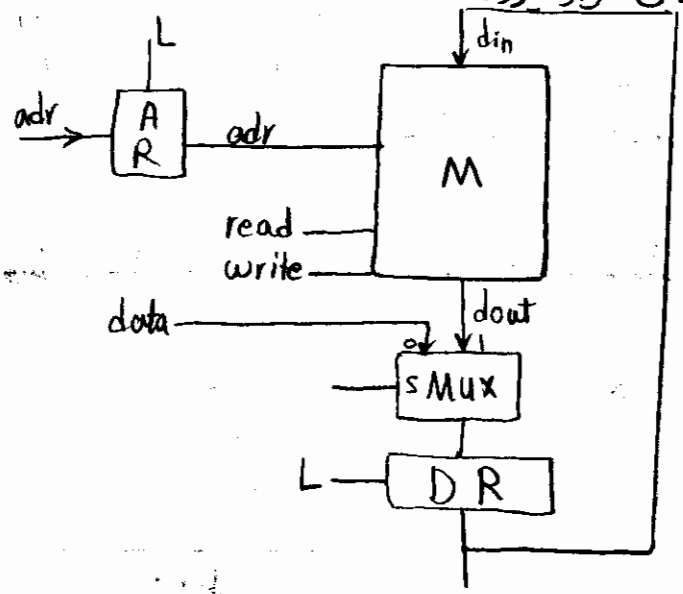
t_d : تأخیر Dec

انتقال با حافظه :



در حافظه مسئله access time زمان دسترسی مطرح می شود که مربوط به تأخیرها می شود.

Read access time : زمانی است که باید مهلت دهیم تا adr و زمان خواندن وارد شود و محتوی در خروجی قرار گیرد.



سیکل خواندن :

T_0 : AR ← adr1

T_1 : DR ← M[AR]

سیکل نوشتن :

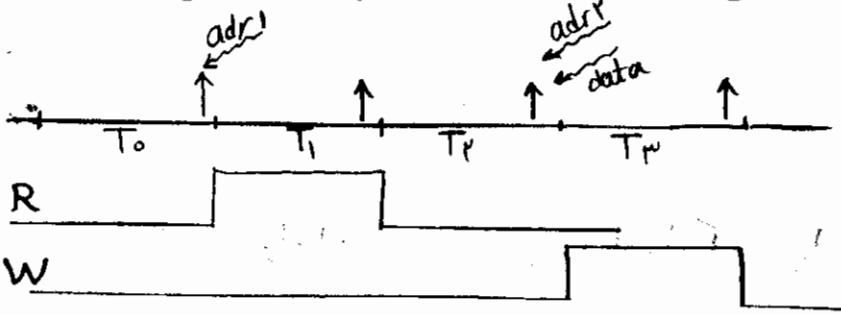
T_2 : AR ← adr2, DR ← data

T_3 : M[AR] ← DR

$L_{AR} = T_0 + T_2$ ، $L_{DR} = T_1 + T_2$

$S = T_1$ ، $Read = T_1$ ، $write = T_3$

در حافظهٔ فوق access time باید کمتر از یک کلاک باشد.



حال اگر پریود کلاک نصف شود باید فرمان ها را مدت بیشتری نگه داریم:

T_0 : $AR \leftarrow adr$

T_1 :

T_2 : $DR \leftarrow M[AR]$

به شرط اینکه access time کمتر از پریود کلاک باشد.

یا

T_0 : $AR \leftarrow adr$

T_1 : $R \leftarrow 1$

T_2 : $DR \leftarrow M[AR]$

همیشه درست

RTL مقابل درست است اما

T_0 : $AR \leftarrow adr1$

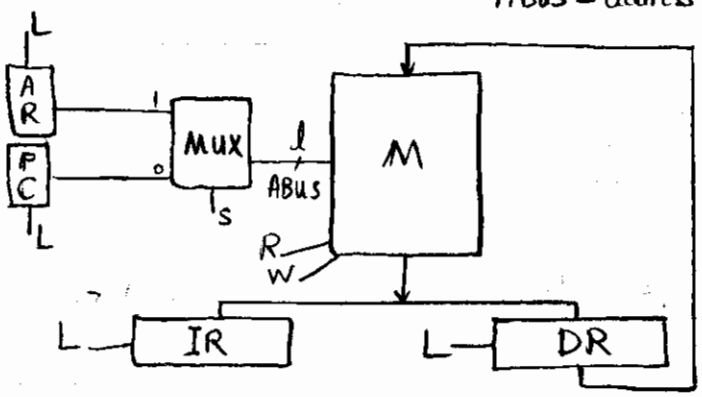
T_1 : $PC \leftarrow adr2$, $DR \leftarrow M[AR]$

T_2 : $AR \leftarrow adr3$, $IR \leftarrow M[PC]$

T_4 : $M[AR] \leftarrow DR$

مدار آن با حافظهٔ قبلی متفاوت است:

ABUS = address Bus



$L_{AR} = T_0 + T_4$ ، $L_{PC} = T_4$ ، $L_{DR} = T_1$ ، $R = T_1 + T_2$

$L_{IR} = T_2$ ، $W = T_3$ ، $S = T_1 + T_3$

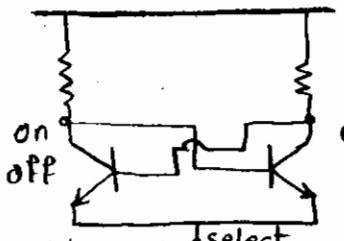
در زمانهای T_1 و T_3 ، AR روی Bus است، به همین دلیل $S = T_1 + T_3$ است.

سلول حافظه:

سلول حافظه F F نیست. سلولهای حافظه دو نوعند استاتیک و دینامیک.

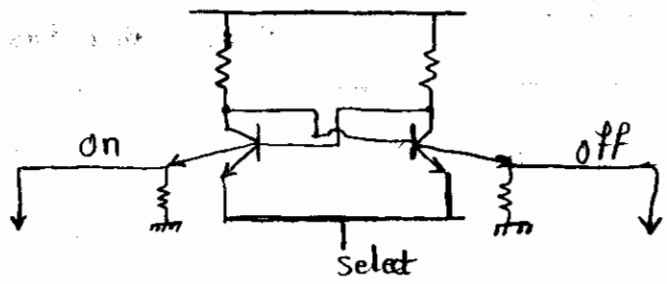
خواندن و نوشتن همزمان در آنها مجاز نیست.

یک سلول پایه به شکل زیر است:



به سلول مقابل سلول استاتیک می گوئیم. وتا وقتی که برق وصل است اطلاعات آن تغییر نمی کند مگر اینکه ما آن را تغییر دهیم.

امّا در سلول دینامیک اطلاعات ممکن است عوض شود.



برای خواندن select را فعال می کنیم و یکی از ترانزیستورها on شده و دیگری off می شود.

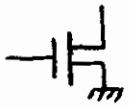
امّا برای نوشتن نه تنها اینکه select را فعال می کنیم امیتر یکی از ترانزیستورها سیگنال

نیز اعمال می‌کنیم. لذا به دلائل مداری:

* خواندن و نوشتن همزمان در سلول حافظه مجاز نیست.

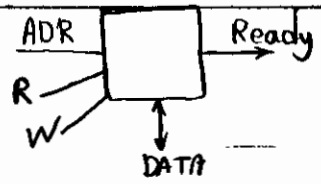
نادرست. ~~M[۱۰] ← R۱ , M[۱۵] ← R۲~~

در سلول دینامیک شارژ را بر روی یک خازن قرار می‌دهیم. لذا بعد از مدتی خازن دشارژ می‌شود و اطلاعات از بین خواهد رفت، این در حالی است که برق هم وصل است.



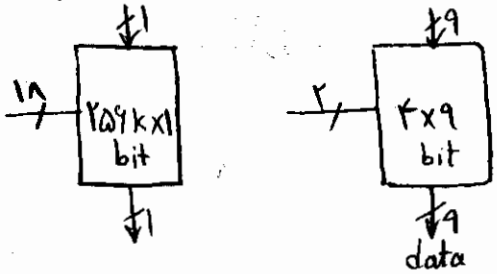
حسن دینامیک این است که به جای چند ترانزیستور یک خازن وجود دارد و لذا در سطح کمتری تعداد بیشتری سلول وجود خواهد داشت. برای رفع عیب این سلول‌ها قبل از اینکه اطلاعات آنها از بین رود، اطلاعات را دوباره تازه می‌کنیم. به این کار Refresh گفته می‌شود. پس یک زمان Refresh مطرح می‌شود. خرکاس مینیمم کلاک را زمان Refresh نیز تعیین می‌کند (علاوه بر ورودی‌های آسنکرون). اگر فرکانس از حد خاصی کمتر شود اطلاعات از بین خواهد رفت.

در سلول‌های دینامیک زمان access time بسته به اینکه سلول در حال Refresh شدن است یا نه متفاوت خواهد بود. لذا یک پایه اضافه می‌شود که اعلام می‌دارد آیا سلول



Refresh شده است یا نه!

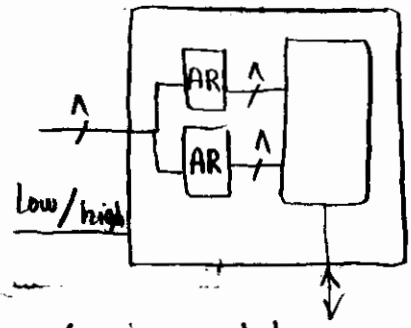
اندازهٔ یک IC وسطحی که اشغال می‌کند بستگی به تعداد پایه های آن دارد . برای یک RAM که دارای ۳۲ کلمه است اگر بخواهیم ۳۲ پایه ورودی و ۳۲ پایه خروجی داشته باشیم ۶۴ پایه



داریم که خیلی زیاد است.

در شکل بالا دو IC با پایه های مساوی و ظرفیت‌های مختلف نشان داده شده است. برای ساخت یک حافظهٔ $512K \times 9$ bit باید IC ۱۲۸۰۰۰ از نوع سمت راست یا IC ۱۸ از نوع سمت چپ داشته باشیم.

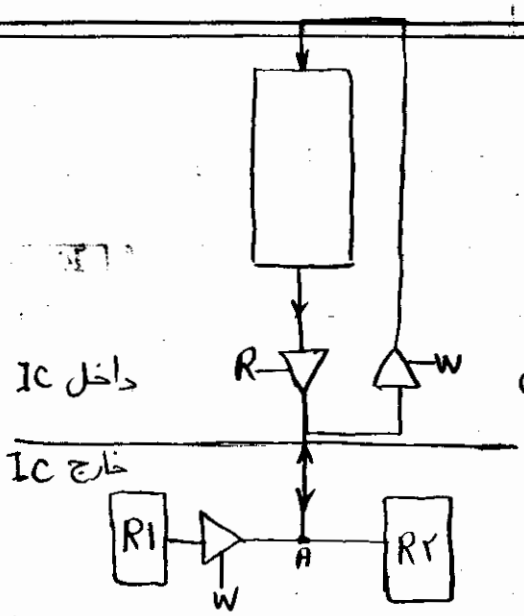
از طرفی می‌توان صرفه جویی پایه کرد. چون خواندن و نوشتن data به طور همزمان مجاز نیست خط data را دو طرفه (BUS) می‌کنیم و یک پایه کمتر می‌شود.



روش دیگر صرفه جویی پایه ها در مقابل نشان داده شده است. به جای ۶ خط آدرس از ۸ خط استفاده شده است و یک low/high مشخص می‌کند کدامیک از AR ها باید وارد شود.

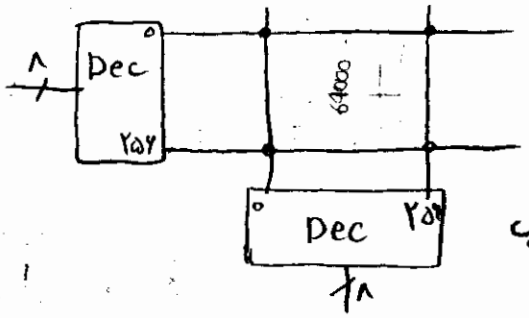
نحوهٔ دوطرفه شدن خط data در

شکل مقابل نشان داده شده است.



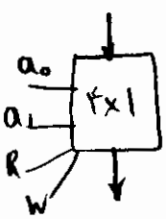
درنقطهٔ A یک BUS برای اطلاعات ورودی وخروجی تشکیل شده است.

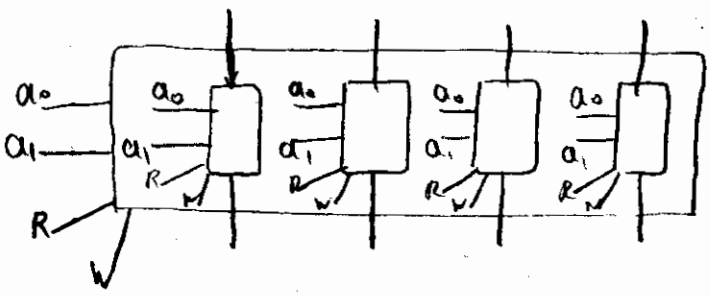
وقتی خط آدرس ۱۶تایی به دوخط ۸تایی تقسیم می‌شود نیاز به یک Dec 16×2^{16} است که یک خط select دارد. لذا نیاز به 2^{16} گیت AND دارد. امّا می‌توان از ترکیب زیر استفاده کرد:

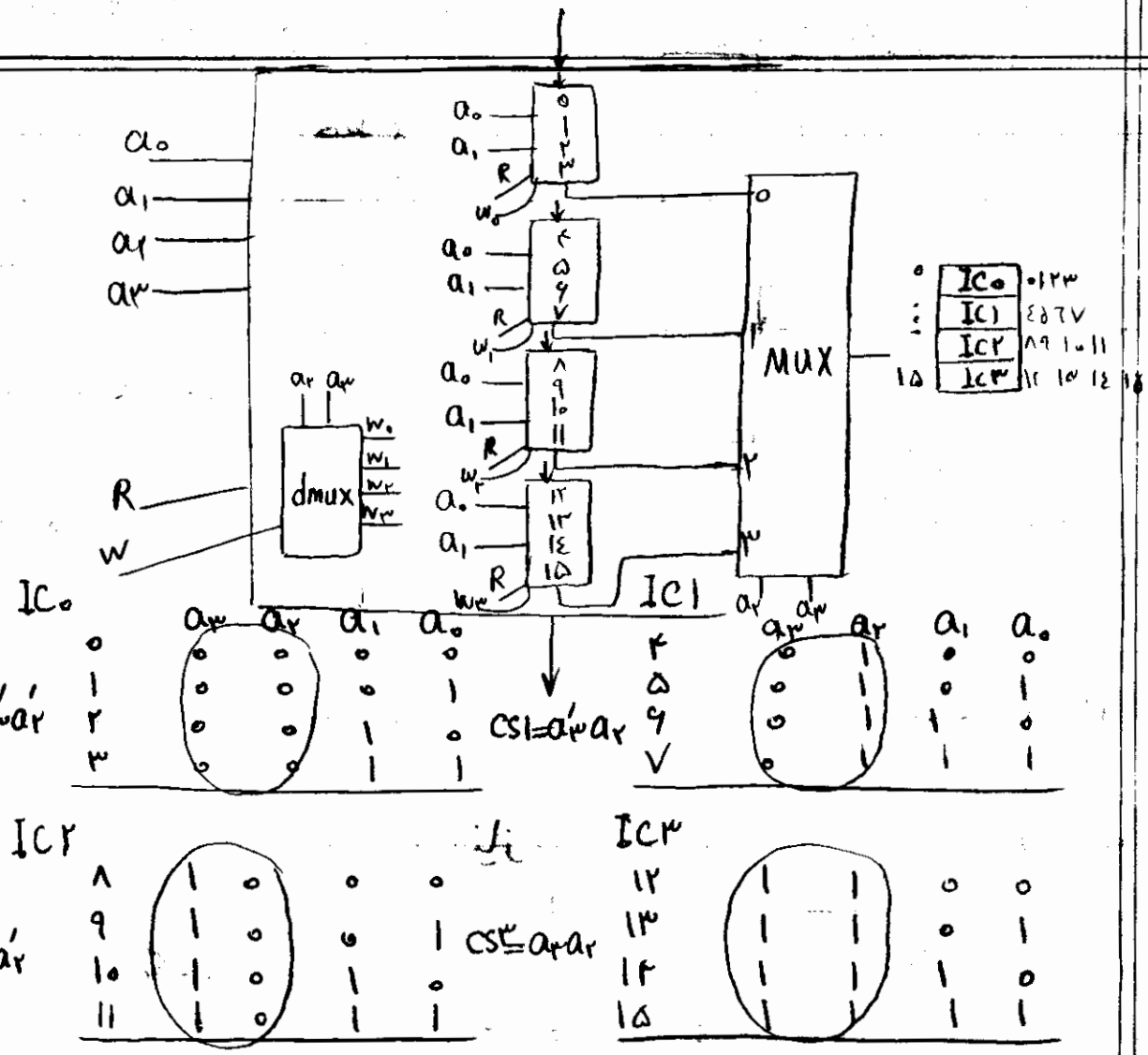


دراین حالت دوخط select داریم. یک کلمه وقتی انتخاب می‌شود که ستون افقی وعمودی آن دارای اطلاعات باشد.

توسعهٔ تعداد کلمات یا تعداد بیت (نقشه حافظه):

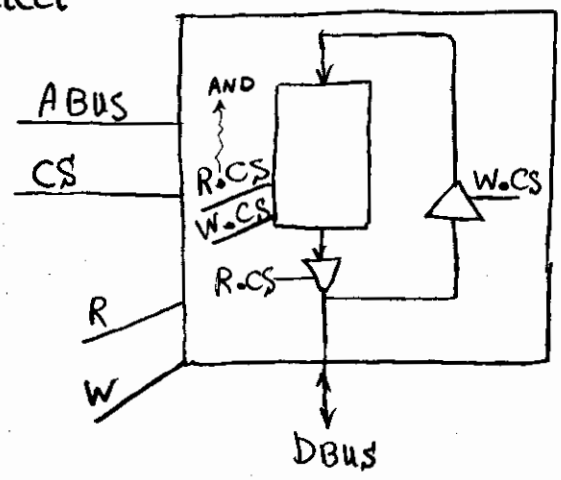


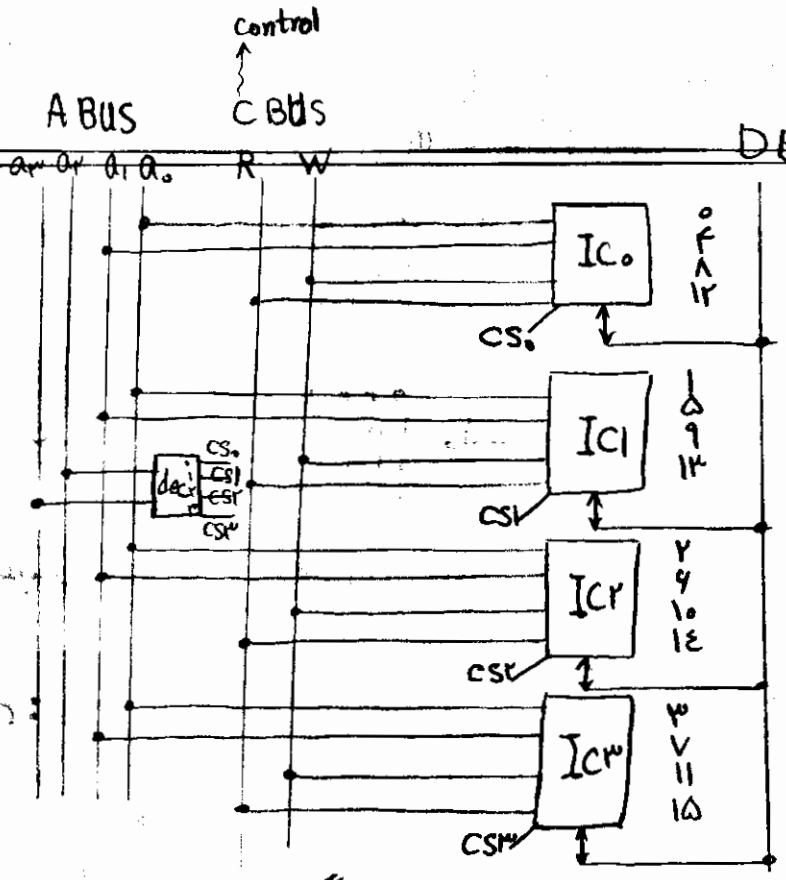




برای اینکه نیازبه MUX, DMUX نباشد از BUS استفاده می کنیم.

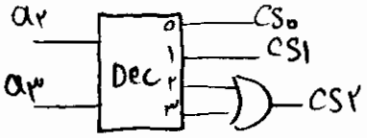
CS = cheap select

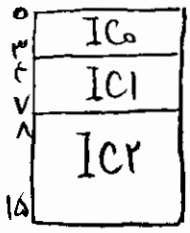




اگر بخواهیم کلمات مشخص شده در بالا در IC ها قرار گیرند کافیست به IC ها و ورودی های a_2 و a_3 و به Dec ورودی های a_0 و a_1 را می دهیم. به این نوع آدرس دهی گسسته اطلاق می شود. آدرس دهی پیوسته در حالت اول بود که در IC_0، ۰،۱،۲،۳، در IC_1، ۴، ۵، ۶، ۷ و ... قرار می گرفت.

آدرس دهی پیوسته:





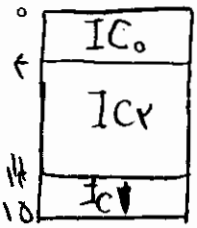
تعویض جای IC ها باعث تغییر CS می شود.

$$CS_0 = a'_3 a'_2$$

$$CS_2 = a'_3 a_2 + a_3 a'_2$$

$$CS_1 = a_3 a_2$$

آدرس دهی پیوسته:



$a_3\ a_2\ a_1\ a_0$

اگر به یک IC کلمات $a_1' a_0$ داده شود cs آن $a_1' a_0$ است.

00	11
01	11
10	11
11	11

اگر به IC کلمات

000	1
101	1
111	0
110	0

داده شود cs آن پیچیده شده و مدارات اضافی خواهیم داشت. لذا ساده‌ترین حالت این است که از آدرس دهی پیوسته استفاده کنیم.

Hex : $a_{15} \dots a_0$

تمرین :

آدرس های ورودی و cs های هر IC را بیابید.

1K	
2K	
1K	
4K	
2K	
8K	

$= 18K$

مثال :

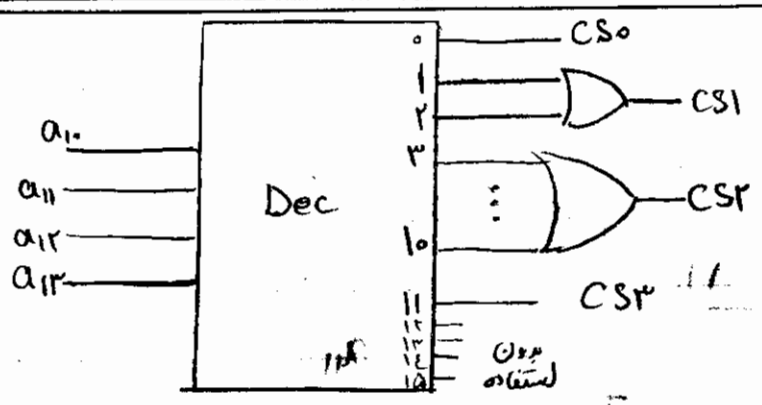
	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	Hex	
Ic0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	0000 ~ 03FF	1K
Ic1	0	0	0	1	x	x	x	x	x	x	x	x	x	x	0400 ~ 0BFF	2K
Ic2	0	0	1	0	x	x	x	x	x	x	x	x	x	x	0C00 ~ 2BFF	4K
	0	0	1	1	x	x	x	x	x	x	x	x	x	x		
	0	1	0	0	x	x									2C00 ~ 2FFF	1K
	0	1	0	1												
	0	1	1	0												
	0	1	1	1	x	x										
	1	0	0	0												
	1	0	0	1												
Ic3	1	0	1	0	x	x										
	1	0	1	1	x	x										

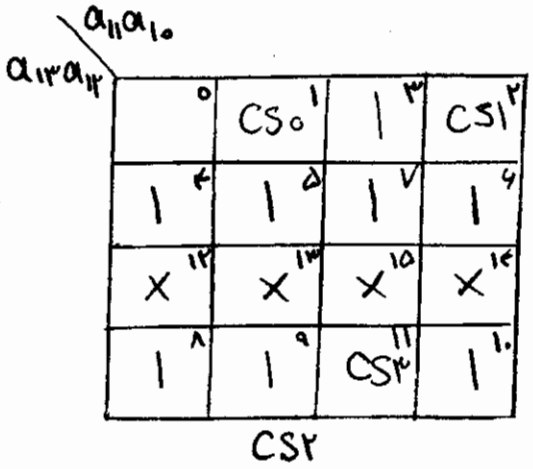
$CS_0 = a_{13}' a_{12}' a_{11}' a_{10}'$

$CS_1 = a_{13}' a_{12}' a_{11}' a_{10} + a_{13}' a_{12}' a_{11} a_{10}'$

$CS_2 =$

$CS_3 =$



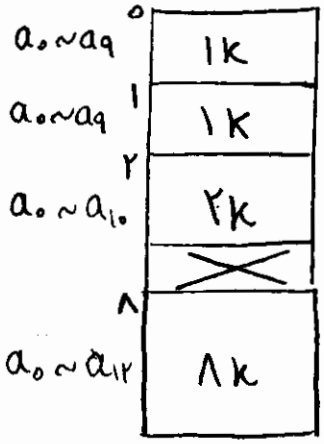


اگر از X ها در ساده سازی CS2 استفاده کنیم دیگر نمی توانیم IC چهاری بعد از IC های قبل استفاده کنیم چون خروجی های ۱۵، ۱۴، ۱۳، ۱۲ Dec را همواره ای میگیریم.

اگر علاوه بر استفاده X ها در ساده سازی CS2 در ساده سازی CS3 هم استفاده کنیم، اگر احیاناً یکی از آدرس ها ۱۲ یا ۱۳ یا ۱۴ یا ۱۵ ۱ شود هم CS3 و هم CS2 فعال می شود. که مشکل ایجاد می کند.

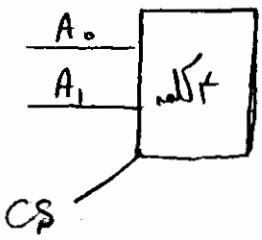
حتماً مجبور نیستیم که آدرس های متوالی را در یک IC قرار دهیم. اگر از آدرس های متوالی استفاده کنیم نه تنها اینکه CS ها تغییر می کند، خطوط وصل شده به IC ها نیز تغییر می کند.

بهترین نقشه برای قرار دادن IC ها این است که هرکدام سرمضربی از مقدار خودش در خطوط آدرس قرار بگیرد.

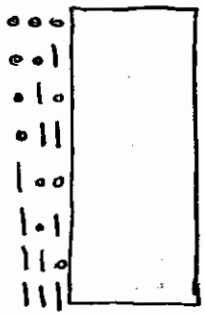


در نقشه مقابل بهترین شکل قرار دادن IC ها ارائه شده است.

حالت پیچیدهٔ انتخاب کلمات:



a_2	a_1	a_0	
0	0	0	m_0
0	0	1	m_1
1	0	1	m_5
1	1	1	m_7



$$CS = a'_2 a'_1 a'_0 + a'_2 a'_1 a_0 + a_2 a'_1 a_0 + a_2 a_1 a_0$$

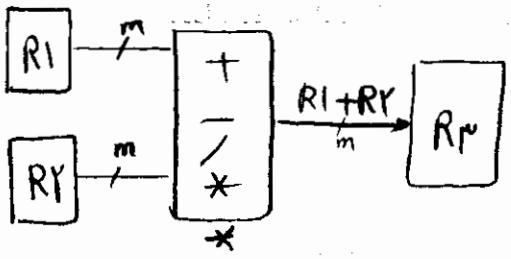
$$A_0 = m_1 + m_7$$

$$A_1 = m_5 + m_7$$

μ-OP های حسابی:

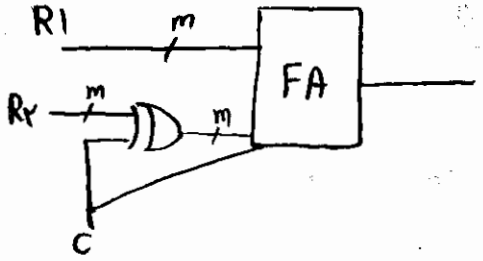
$R3 \leftarrow R1 + R2$
$R3 \leftarrow R1 - R2$
$R3 \leftarrow R1 * R2$
$R3 \leftarrow R1 / R2$

$R \leftarrow R+1$
$R \leftarrow R-1$
$R \leftarrow Const$
$R \leftarrow -R$



با توجه به اینکه μ-op باید در یک کلاک انجام شود لذا مدار داخل * باید یک مدار ترکیبی باشد که در یک کلاک انجام می‌شود.

μ-op تفریق :

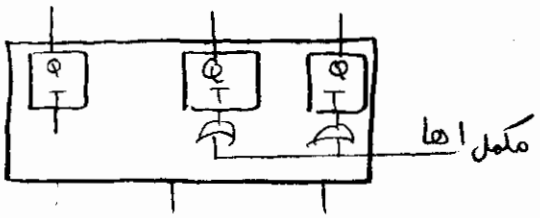


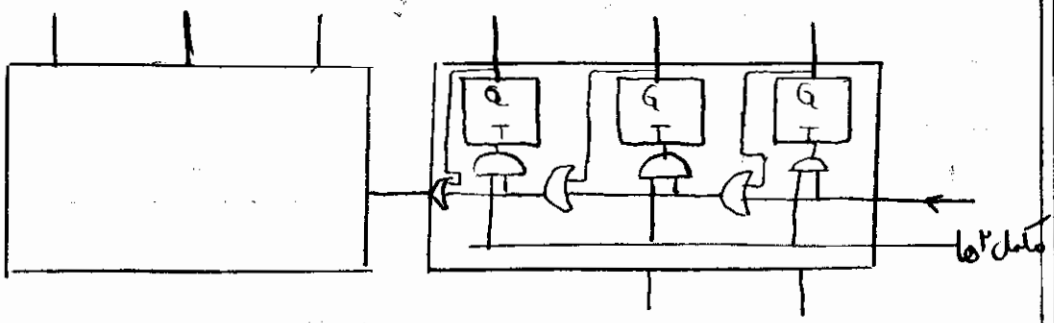
داشت.

می توان ۴ μ-op جمع و تفریق و ضرب و تقسیم را به یک MUX داد و همگی را در یک زمان

برای $\bar{R} \leftarrow -R$:

$$R \leftarrow -R \equiv \begin{cases} \bar{R} \leftarrow \bar{R} \\ R \leftarrow R+1 \end{cases}$$





μ-op های منطقی :

$R۳ \leftarrow R1 \wedge R۲$ selective clear
$R۳ \leftarrow \bar{R۳}$
$R۳ \leftarrow R1 \vee R۲$ selective set
$R۳ \leftarrow ۰$
$R۳ \leftarrow R1 \oplus R۲$ selective Complement
$R۳ \leftarrow$ ۱ یا $\bar{۰}$
$R۳ \leftarrow R1 \odot R۲$

μ-op ها ی منطقی عملیات پایه‌ای هستند که برای پردازش غیرعددی و تست ها بکار می روند.

مثال : می‌خواهیم دو بیت a_2 و a_7 از ترکیب زیر صفر شوند.

$$R = \begin{array}{|cccccccc|} a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\ \hline 0 & a_6 & a_5 & a_4 & a_3 & 0 & a_1 & a_0 \end{array}$$

$\longrightarrow R \wedge (\underbrace{01111011}_{\text{mask}})$: selective clear

AND شدن در این نوع μ-op ها به صورت بیت به بیت است (و به همین ترتیب دیگر اپراتورها). فلذا $R1 + R2$ (حسابی) ، $R1 \vee R2$ (منطقی) باهم متفاوتند.

البته اگر توابع بولی باشند (مانند توابع هرمان) شکلهایی مانند $T_1 + T_2$ به مفهوم منطق است و ایرادی ندارد.

$R \vee (10000100)$: selective set

$R \oplus (10000100)$: selective complement

در یک کامپیوتر جمع و تفریق و... بر روی اطلاعات BCD یا BIN انجام می گیرد.

۹۲ : $\underbrace{\underbrace{0011\ 1001}_{9\ ASCII}\ \underbrace{0011\ 0010}_{2\ ASCII}}_{92\ ASCII}$

$\underbrace{\underbrace{00001001}_{9\ BCD}\ \underbrace{00000010}_{2\ BCD}}_{92\ BCD\ unpacked}$

UNP

۹۲ BIN ← 1011100

۹۲ BCD packed ← 1001 0010

در مورد حروف :

A 0100 0001
B 0100 0010
⋮
a 0110 0001
b 0110 0010
⋮

if R حرف then

حرف بزرگ: R∧(1101 1111)

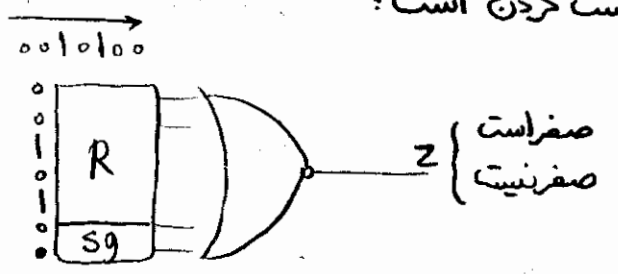
حرف کوچک: R∨(0010 0000)

تغییر: R⊕0010 0000

روش دیگر تبدیل حروف:

if 'a' ≤ R ≤ 'z' then R-32

استفاده دیگر OP-M های منطقی در تست کردن است:



بالاترین بالا

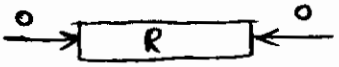
R∧(0000001) ← مثبت و منفی بودن عدد

⋮ ⋮

برای تشخیص اینکه مثلاً آیا بیت ۲ و ۴ صفر و بیت ۳ یک است می توانیم از دو OP-M منطقی استفاده کنیم و در دو مرحله تست را انجام دهیم.

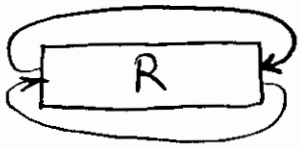
OP-M های شیفت:

شیفت منطقی : shr R , shl R



در شیفت منطقی به چپ و راست عنصر جایگزین شده صفر است.

Circular shift : Cir R , Cil R



شیفت حسابی
Arith. shift :

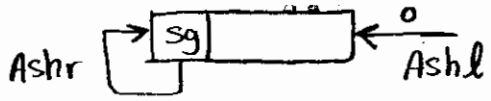
Ashr R , AshlR

Ashr R
تقسیم بر ۲
با رعایت علامت

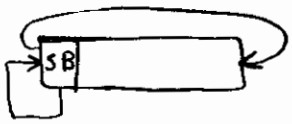
AshlR
ضرب در ۲
با رعایت علامت

1000 ... 0

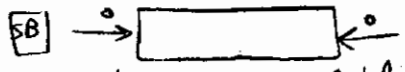
اگر قرارداد نمایش مکمل ۲ باشد :



" " " مکمل ۱ باشد :



" " " علامت وقدرمطلق باشد :



Ashr R
تقسیم بر ۲
با رعایت علامت.

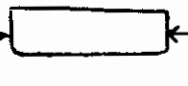
Ashl R
ضرب در ۲
با رعایت علامت

مثال علامت وقدرمطلق :

%۱	۰۱۱۰	±۶	R
%۰	۱۱۰۰	±۱۲	Ashl R
%۱	۰۰۱۱	±۳	Ashr R

مثال مکمل ۱ :

+۶	۰۰۱۱۰	۰ +
+۳	۰۰۰۱۱	۱ −
+۱۲	۰۱۱۰۰	



در مکمل ۱ برای شیفت اعداد منفی نخست آنها را مثبت کرده و به روش مثبت شیفت می‌دهد وسپس تبدیل به منفی می‌کند.

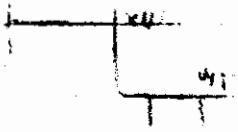
منفی $a_4\ a_3\ a_2\ a_1\ a_0$
مثبت $a'_4\ a'_3\ a'_2\ a'_1\ a'_0$

$a'_3\ a'_2\ a'_1\ a'_0\ 0 \longrightarrow a_3\ a_2\ a_1\ a_0\ 1$

$0\ a'_4\ a'_3\ a'_2\ a'_1 \longrightarrow 1\ a_4\ a_3\ a_2\ a_1$

پس در این حالت علامت هر چه باشد وارد می‌شود Ashl R= مکمل۱ Cil R

در مکمل ۲ :

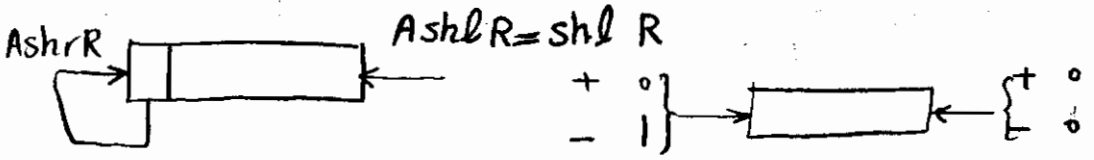


منفی $a_5\ a_4\ a_3\ 1\ 0\ 0$

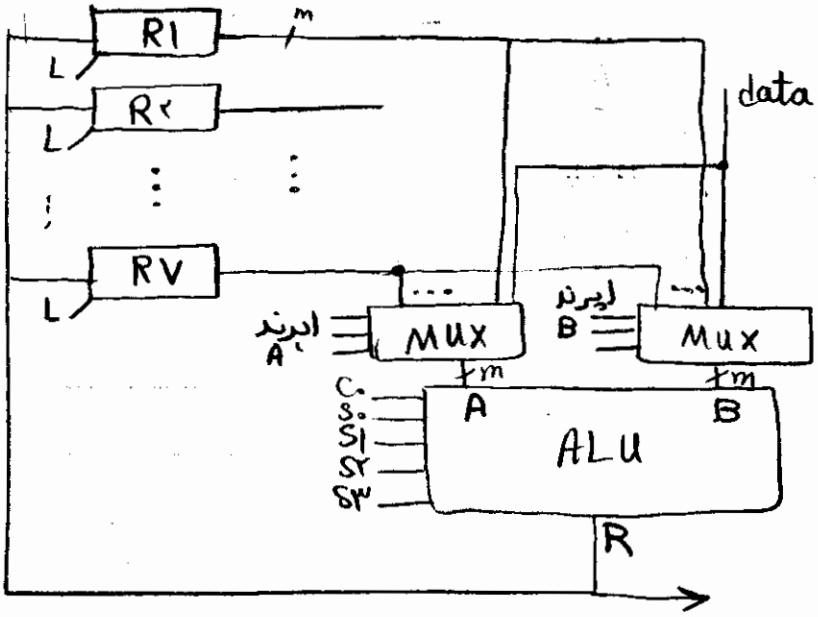
مثبت $a'_5\ a'_4\ a'_3\ 1\ 0\ 0$

$a'_4\ a'_3\ 1\ 0\ 0\ 0 \longrightarrow a_4\ \bar{a}_3\ 1\ 0\ 0\ 0$

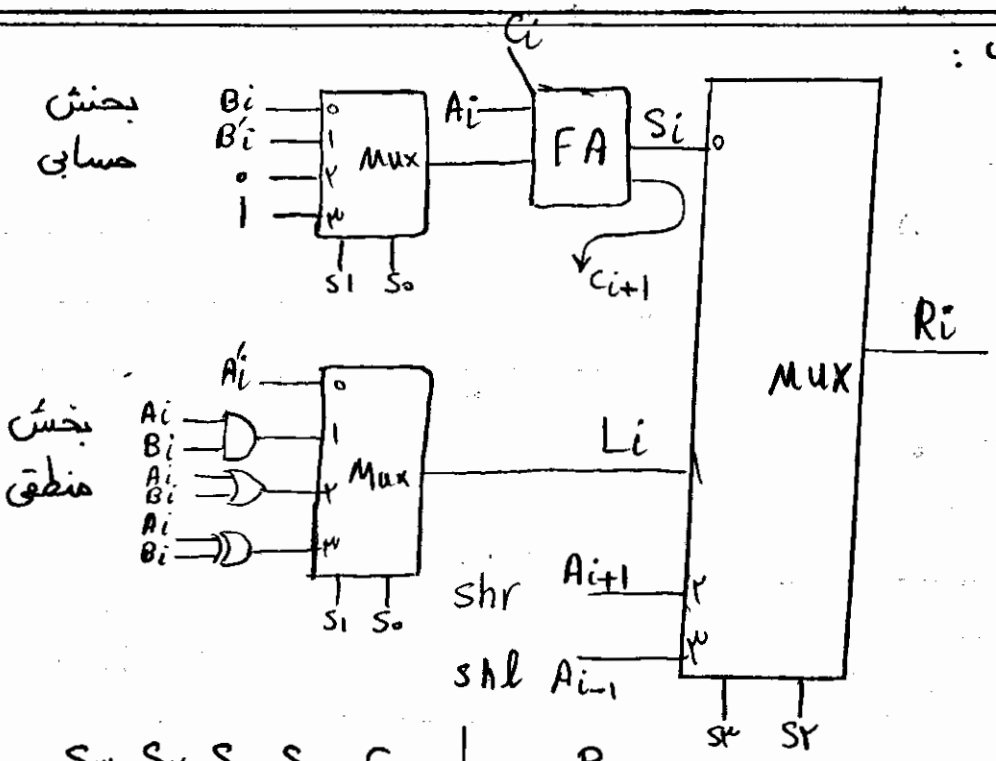
$0\ a'_5\ a'_4\ a'_3\ 1\ 0 \longrightarrow 1\ a_5\ a_4\ a_3\ 1\ 0$



می‌خواهیم تمام μ-OP ها را در مجموعهٔ زیر داشته باشیم. برای این منظور یک مدار ترکیبی به نام ALU خواهیم داشت و رجیسترها فقط قابلیت بارگیری دارند.



برای یک بیت :



S_3	S_2	S_1	S_0	C_0	R	
0	0	0	0	0	A+B	ADC
		0	0	1	A+B+1	ADC
		0	1	0	A−B−1	SBB
		0	1	1	A−B	SBB
		1	0	0	A	
		1	0	1	A+1	
		1	1	0	A−1	
		1	1	1	A	
0	1	0	0	x	$\bar{A}$	
0	1	0	1	x	A∧B	
0	1	1	0	x	A∨B	
0	1	1	1	x	A⊕B	
1	0	x	x	x	shr A	
1	1	x	x	x	shl A	

حسابی

منطقی

$+A_n \cdots A_0$ → 1 1 1 1 D → -1

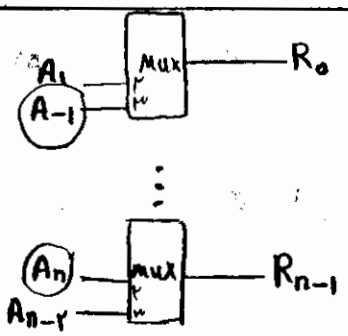
ADD : A+B

ADC : A+B+Carry (برای جمع قبلی)

SUB : A−B

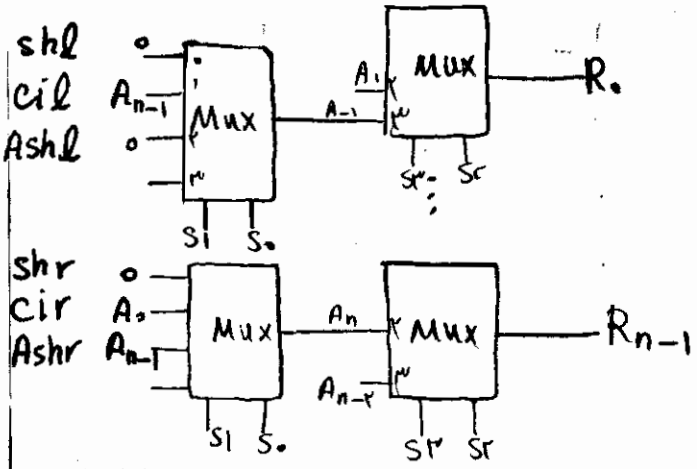
SBB : A−B−borrow (برای تفریق قبلی)

درمورد شیفت :



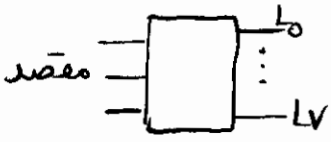
آنچه که نوع شیفت رامشخص می کند A_{-1} , A_n است. لذابرای انتخاب انواع

شیفت از Mux استفاده می کنیم :



S۳	S۲	S۱	S۰	C۰	R
۱	۰	۰	۰	α	shr A
		۰	۱	α	cir A
		۱	۰	α	Ashr A
۱	۱	۰	۰	α	shl A
		۰	۱	α	cil A
		۱	۰	α	Ashl A

برای load رجیسترها از dec استفاده می کنیم :



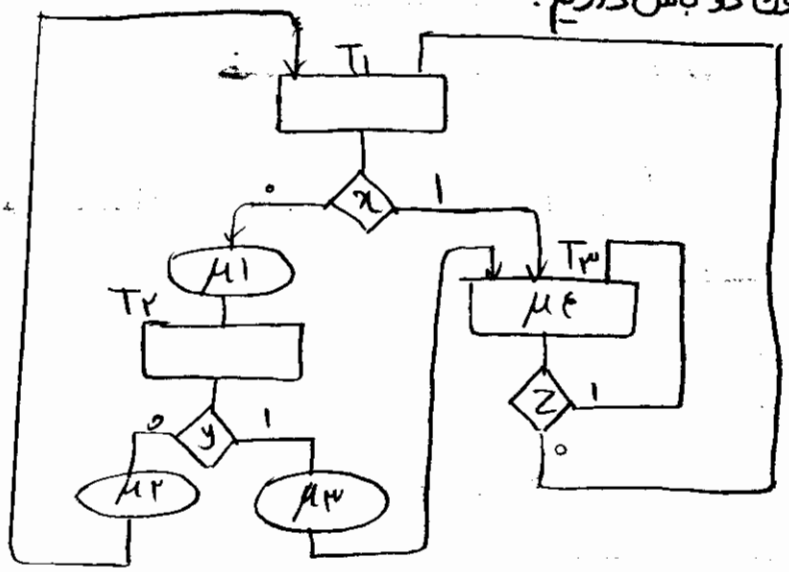
لذا کلمهٔ واحد کنترل ۱۴ بیتی خواهد بود :

اپرند A		اپرند B		مقصد			=
۳	+	۳	+	۳ (برای load ها)	+	۵ (S۳ S۲ S۱ S۰ C۰)	۱۴

در این ALU صفر کردن یک رجیستر توسط XOR کردن رجیستر با خودش انجام می‌شود.

خروجی های A و B در ALU باس هستند. همچنین نمی‌توان با خروجی سه حالته آنها

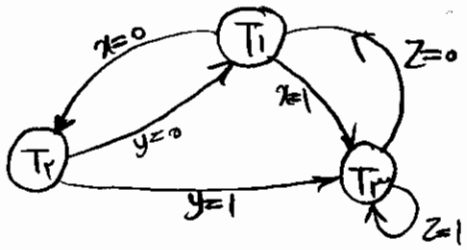
را تشکیل داد چون دو باس داریم.



$A'B'$ T_1 : if x' then (μ_1, goto T_2) else (goto T_3)

$A'B$ T_2 : if y' then (μ_2, goto T_1) else (μ_3, goto T_3)

AB' T_3 : μ_4, if z' then (goto T_1) else (goto T_3)



روش دیگر نشان دادن : A, B فلیپ فلاپ هستند که T_1 تا T_3 را تولید می‌کنند.

$A'B'x'$: μ_1 , $B \leftarrow 1$

$A'B'x$: $A \leftarrow 1$

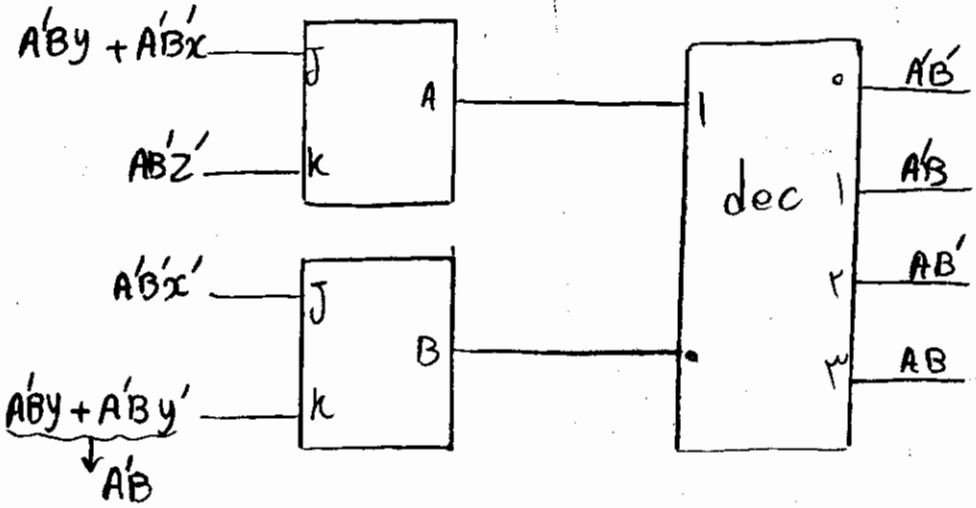
$A'By'$: μ_2 , $B \leftarrow 0$

$A'By$: μ_3 , $A \leftarrow 1$, $B \leftarrow 0$

$AB'Z'$: $A \leftarrow 0$

$AB'Z$: ——— نوشتن این سطر در RTL نیازی نیست.

با اینکه دنبال کردن RTL مشکل است امّا این حسن را دارد که مدار کنترل را از روی آن به راحتی می‌توان ساخت.

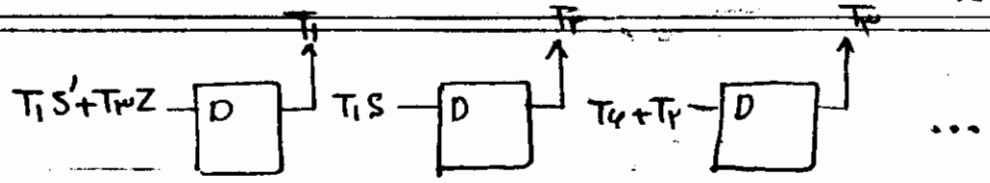



```
10  if s then Read A, B else goto 10
    P = 0
20  if B=0 then goto 10
    P = P + A
    B = B - 1
    goto 20
```

حالت فعلی	μ-Op	تغییر حالت
T_1	if s then $RA \leftarrow A$, $RB \leftarrow B$	if s' then goto T_1 else T_2
T_2	$RP \leftarrow 0$	goto T_3
T_3	———	if (Z) ($RB=0$) then goto T_1 else T_4
T_4	$RP \leftarrow RP + RA$	goto T_5
T_5	$RB \leftarrow RB - 1$	goto T_4
T_6	—	goto T_3

پردازشگر و توابع کنترلی — واحد کنترل

طراحی با یک فلیپ فلاپ برحالت :



نهش حالتها :

مرحلهٔ اول:

T_1	if s then $RA \leftarrow A$, $RB \leftarrow B$, $RP \leftarrow 0$	if s' then T_1 else T_2
T_2	—	if z then T_1 else T_3
T_3	$RP \leftarrow RP+RA$, $RB \leftarrow RB-1$	T_2

مرحلهٔ دوم :

T_1	if s then $RA \leftarrow A$, $RB \leftarrow B$, $RP \leftarrow 0$	if s' then T_1 else T_2
T_2	if z' then $RP \leftarrow RP+RA$, $RB \leftarrow RB-1$	if z then T_1 else T_2

تکلیف۲ : ۴ فصل ۴ مانو ۶، ۱۲، ۱۳، ۱۴، ۱۵، ۱۸، ۱۹، ۲۰، ۲۱

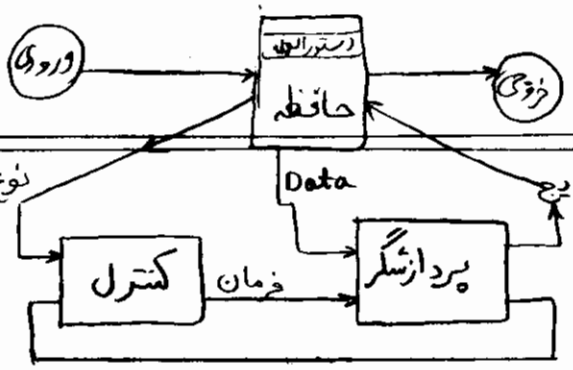
+ مدار تقسیم با if else goto

+ " " " بدون " "

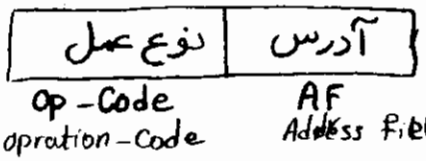
+ باس سری برای چهار رجیستر

+ رجیستر R با JkFF و قابلیتهای

Load → $R \leftarrow Input$
$R \leftarrow 0$
$R \leftarrow -1$
$R \leftarrow IAR$
$R \leftarrow IVR$
$R \leftarrow I \oplus R$



فرمت دستورالعمل



تعداد آدرس موردنیاز بستگی به نوع عمل دارد :

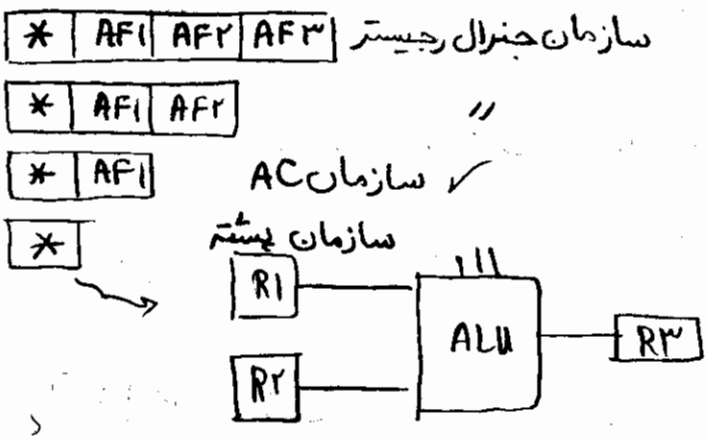
دستور یونری (یک اپرندی) مانند not . ۲ آدرس لازم داریم.

دستور باینری (دو اپرندی) مانند X یا / ۳ آدرس " ".

دستور انتقال (امبدا + امقصد) ۲ آدرس "

کنترل غیرمشروط ۱ " "

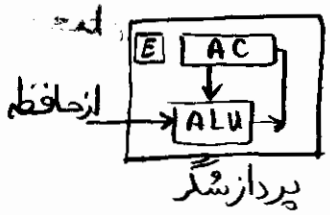
کنترل مشروط . ۲ " "



هرجا که آدرس کم است به این معنی است که آن آدرس درسخت افزار وجود دارد و

مقید است که آن آدرس را انجام دهد. کامپیوترهای اولیه به طریق سازمان AC بودند

و برای عملی مانند ضرب فقط یک AF نیاز داشتند.



در این سازمان یک ایرند از حافظه و دیگری از AC گرفته می‌شود و نتیجه در خود AC

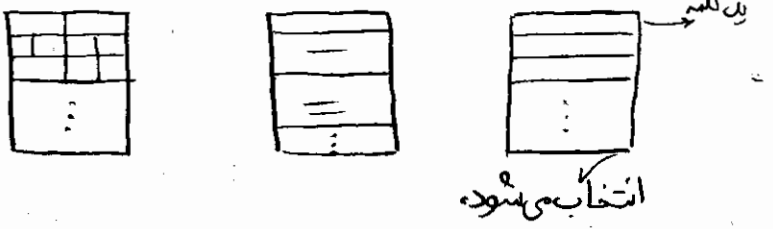
قرار می‌گیرد. E بیت carry است که در جمع و... بکار گرفته می‌شود.

رابطهٔ طول دستور و طول کلمه :

یک روش این است که همهٔ دستورها یک طول دارند و هر کلمه یک دستور است. در روش

دیگر دستورها مضربی از طول کلمه هستند. در روش سوم طول دستورها نیز متفاوت

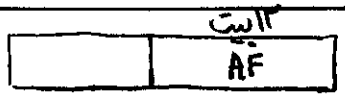
است.



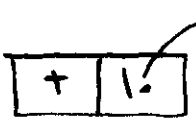
طول کلمات نوعاً مضرب طول کاراکتر است و با توجه به اینکه طول یک کاراکتر ۸ بیت است

لذا طول کلمه مضرب ۸ است.

اندازهٔ حافظه را که شامل کلمات است ۴ کیلوکلمه (4k) درنظر می‌گیریم.



مدهای آدرس دهی :



تفاسیر ۱۰

۱- مد ضمنی — $AC \leftarrow AC + 10$

۲- مد بلافصل — $AC \leftarrow AC + M[10]$

۳- حافظه‌ای مستقیم — $AC \leftarrow AC + M[M[10]]$

۴- حافظه‌ای غیرمستقیم — $AC \leftarrow AC + R_{10}$

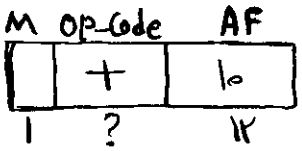
۵- رجیستری مستقیم — $AC \leftarrow AC + M[R_{10}]$

۶- رجیستری غیرمستقیم

در ضمنی آدرس و AF نداریم محل اپرند به صورت ضمنی در سخت افزار مشخص شده است. در جمع نشان داده شده در بالا یک آدرس واضح و دوتای دیگر به طور ضمنی مشخص شده‌اند.

هدف از این تنوع مدها بالابردن راندمان یا سرعت و یا ایجاد تسهیلات است.

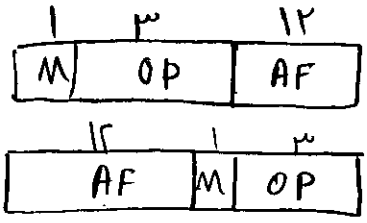
مدهای ۱ و ۳ و ٤ انتخاب می‌شوند. حال مدهای ۳ و ٤ را داریم می‌خواهیم ببینیم باید با کدامیک عمل کنیم. برای این منظور یک بیت به نام Mode اختیار می‌کنیم.



طول op-code هم مضرب بایت است. کامپیوتری که می‌خواهیم تعریف کنیم دارای ۲۵ دستور است. لذا طول کلمه را ۲ بایت می‌گیریم. در نتیجه طول op-code ۳ بیت خواهد بود.

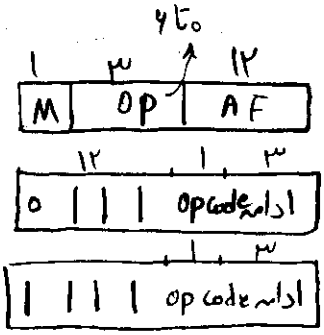
برای اینکه ۲۵ دستور را جای دهیم اینگونه در نظر می‌گیریم که همه دستورها چنین ساختاری

ندارند.



دو ساختمان با لا را می‌توانیم داشته باشیم ولی تشخیص این دو از هم ممکن نیست.

دو ساختمان زیر را انتخاب می‌کنیم:



دستورات:

	دستورالعمل	کد دستور Hex			اثر دستور
پردازشی	AND	۰ یا ۸	۰	۰	AC ← AC ∧ M[EA]
پردازشی	ADD	۱ یا ۹	۰	۱	AC ← AC + M[EF]
انتقال	LDA	۲ یا A	۰	۱	AC ← M[EA]
انتقال	STA	۳ یا B	۰	۱	
کنترل برنامه	BUN	۴* یا C*	۱	۰	PC ← EA = AF* یا M[AF]*
RET, call	BSA	۵ یا D	۱	۱	M[EA] ← PC PC ← EA + ۱
پردازش + کنترل	ISZ	۶ یا E	۱	۱	M[EA] ← M[EA] + ۱ if M[EA] = ۰ then PC ← PC + ۱
پردازشی	CLA	۷	۸	۰	AC ← ۰
	CMA	۷	۴	۰	AC ← $\overline{AC}$
	CLE	۷	۲	۰	E ← ۰
	CME	۷	۱	۰	E ← $\overline{E}$
	INC	۷	۰	۸	AC ← AC + ۱
	cir	۷	۰	۴	cir E, AC
	cil	۷	۰	۲	cil E, AC
	SZA	۷	۰	۱	if AC = ۰ then PC ← PC + ۱
	SPA	۷	۰	۰	۸ if (AC > ۰) then PC ← PC + ۱
	SNA	۷	۰	۰	۴ " AC < ۰ " "
	SZE	۷	۰	۰	۲ " E = ۰ " "
	HLT	۷	۰	۰	۱ توقف

۱- پردازشی

۲- انتقال

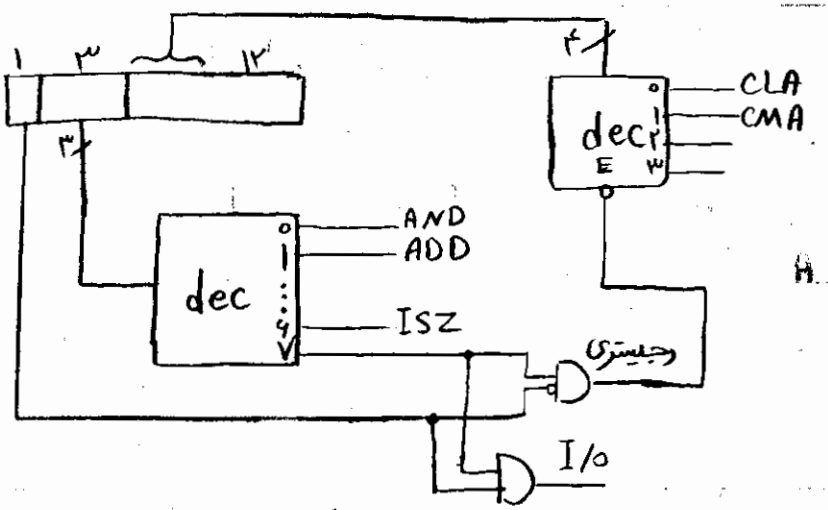
۳- I/O

۴- کنترل پردازنده

۵- کنترل ماشین

	دستور العمل	
I/O	INP	F ۸۰۰
I/O	OVT	F ۴۰۰
کنترل برنامه برای I/O	SKI	F ۲۰۰
کنترل برنامه برای I/O	SLO	F ۱۰۰
کنترل ماشین	ION	F ۰۸۰
کنترل ماشین	IOF	F ۰۴۰

دستورات I/O



با توجه به اینکه دستورات رجیستری و I/O دارای کد ۱۲ بیتی هستند و ۴ بیت استفاده کرده ایم و بقیه تلف می‌شوند لذا دیگر از dec استفاده نمی کنیم و از هر ۱۲ بیت استفاده می کنیم:

اثر دستور AND: $AC \leftarrow AC \wedge M[AF]$ مستقیم

$AC \leftarrow AC \wedge M[M[AF]]$ غیرمستقیم

آدرس موثر EF آدرسی از حافظه است که اپرند مستقیم از آنجا برداشته می‌شود.
effective Add.

موثر $EA = \begin{cases} AF & \text{مستقیم} \\ M[AF] & \text{غیرمستقیم} \end{cases}$

محیط کار برنامه نویس

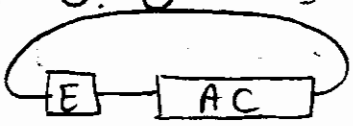
حافظه ای در اختیار دارد.

اثر دستورها بر AC و E باید مشخص شود (برای برنامه نویس)

PC (program counter) باید مشخص شود.

load AC LDA	X	انتقال X به AC از حافظه
store AC STA	Y	انتقال AC به Y از حافظه
LDA STA	A B	انتقال از خانه A حافظه به خانه B از حافظه
LDA ADD STA	A B C	≡ C ← A + B جمع
LDA CMA INC ADD STA	B A C	≡ C ← A − B تفریق

و به همین ترتیب دیگر عملیات منطقی قابل انجام است.

cir یا cil ≡ 

LDA CLE cir یا cil	X	≡ شیفت منطقی

دیگر شیفت ها هم با دوبار لود کردن در AC و cil یا cir قابل انجام است.

دستورات کنترل برنامه ترتیب اجرا را برای ما تعیین می کند.

کامپیوتر از وقتی که روشن می شود مرتب در حین انجام دستورات است و مهم ترتیب اجرای این دستور العمل ها است.

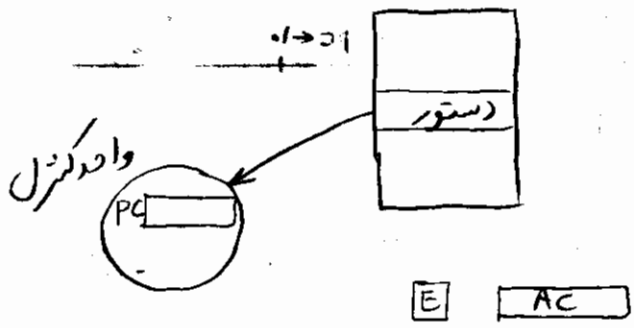
ترتیب اجرای دستور العملها:

Instruction

۱- نقطهٔ شروع که قراردادی است. این نقطهٔ شروع نقطهٔ صفر است. (I_0)

۲- توالی که قراردادی است و لازم نیست که بعد از هر دستور آدرس دستور بعدی را که باید اجرا شود مشخص کنیم.

۳- دستور کنترل برنامه توالی را به هم می زند.



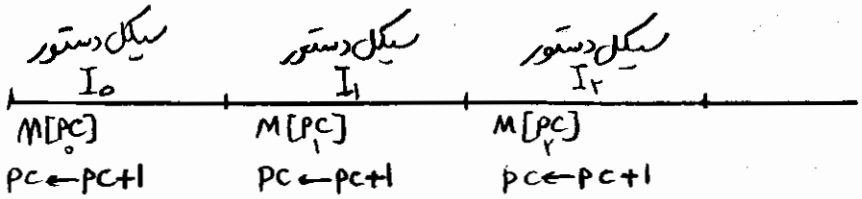
واحد کنترل دستورات را از حافظه برمی دارد و به دیگر قسمتها فرمان می دهد.

PC رجیستری است که شمارهٔ دستوری را که باید اجرا شود در آن وجود دارد. لذا واحد کنترل با توجه به شماره ای که در PC است دستورات را به ترتیب اجرا می کند. لذا در شروع کار

program Counter

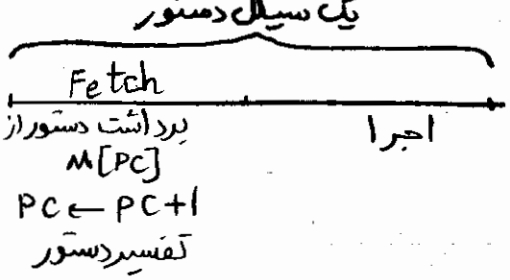
مقدار اولیهٔ PC صفر است. پس واحد کنترل دستور M[PC] که همان M[0] است را از حافظه

می‌برد.



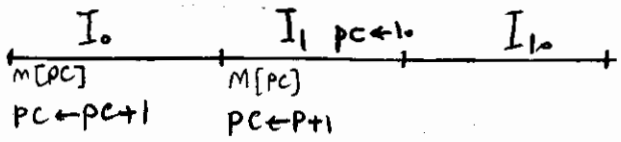
هر بار که یک دستور را از حافظه برداشته می‌شود یک واحد به PC اضافه می‌شود. این شکل

توالی وقتی عوض می‌شود که PC عوض شود.



دستورات کنترلی دستوراتی هستند که در مرحلهٔ اجرا PC مشروط و یا غیرمشروط عوض می‌شود.

نمونه:



همان طور که گفتیم در اثر دستورات زبان برنامه نویس گفته می‌شود. این ها همه در محیط

برنامه‌نویسی باید باشند. محیط برنامه نویسی

دستورات و اثر دستورات

PC

AC, E

حافظه

Branch unconditional

دستور BUN دستوری است که در آن PC برابر EA می‌شود PC ← EA

BUN adr ; pc ← adr

این دستور یک دستور کنترلی غیرمشروط است.

دستور (کدآن)	اثرآن
F100	PC ← 100
C100	PC ← M[100]

Branch Conditional

دستور BCD یک دستور کنترلی مشروط است و برای متوقف سیکل دستورات هم استفاده می‌شود).

BCD adr1 , adr2

یعنی : if (CD) then pc ← adr1 else pc ← adr2

نوعاً برای اینکه طول دستور دو برابر نشود (چون دو adr داریم) لذا به شکل زیر اجرا می‌شود:

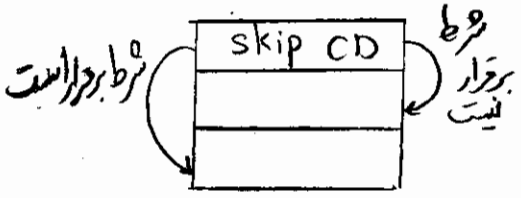
BCD adr1
BUN adr2
≡ BCD adr1 , adr2

در کامپیوترها فرم بالا را می‌بینیم. در این فرم adr1 به شکل صریح گفته می‌شود و adr2 به شکل ضمنی. در حالت اول هر دو adr صریح بود. در دستورات زیر هر دو adr ضمنی خواهند بود :

SPA , SNA , SZA , SZE

اثر این دستور :

if CD then pc ← pc+1

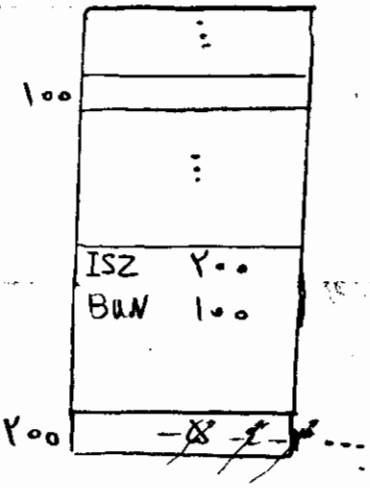


باید دقت شود که در مرحلهٔ Fetch، PC یکی اضافه شده و وقتی شرط برقرار است یک واحد دیگر نیز به آن اضافه می‌شود.

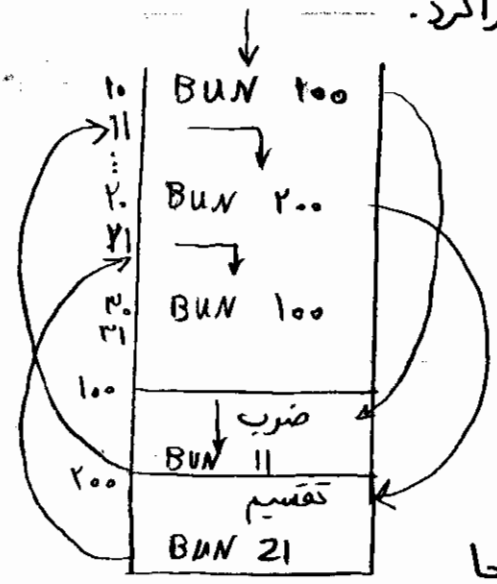
دستور ISZ (Increment skip zero)

ISZ

اثر
$$\begin{cases} M[EA] \leftarrow M[EA]+1 \\ \text{if } M[EA]=0 \text{ then } PC \leftarrow PC+1 \end{cases}$$

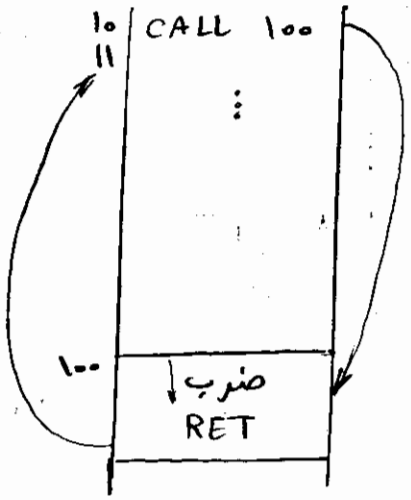


دستورهای CALL و RET (Return call) که برای زیربرنامه‌نویسی به کار می‌رود. وجود زیربرنامه‌ها ضروری نیست (در تئوری) امّا در عمل نمی‌توان برنامه‌های حجیم (با تعداد خط‌های زیاد) را بدون زیربرنامه‌ها پیاده و اجرا کرد.



مشاهده می‌کنیم که تکلیف مقابل نمی‌تواند مقدار زیربرنامه را داشته باشد. چون بعد از رفت دیگر به همان جایی که از آنجا آمده بر نمی‌گردد. (که در BUN ۱۰۰ واقع در خط ۳۰ این امر مشاهده می‌شود)

اما در CALL می‌رود و به همانجا (با استفاده از RET) برمی‌گردد.



لذا آدرس برگشت نیز باید در دست باشد. لذا دو آدرس یکی برای رفت و دیگری برای برگشت نیاز داریم. هنگامی که می‌رود آدرس برگشت ضبط (Save) می‌شود.

دستور	اثر آن	
CALL	هر دو انجام می‌شود.	برای برگشت ← ضبط PC
		برای رفت ← مقداردهی به PC

لذا آدرس برگشت به صورت ضمنی خواهد بود و خود مدار می‌گوید که آدرس کجا باید ضبط شود. محل ضبط می‌تواند رجیستر، پشته حافظه، انتخاب کتاب باشد

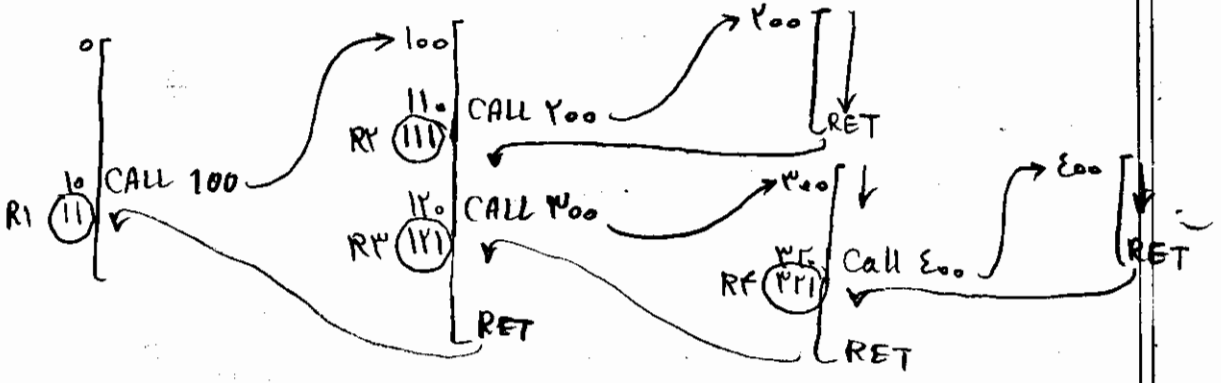
در این حالت (حالت یک رجیستر RAR) نمی‌توان

رجیستر Repair adress register

CALL { RAR ← PC ; PC ← EA }

RET PC ← RAR

CALL, RET تودرتو داشت.



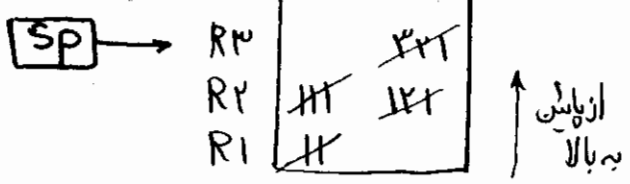
last in first out

روشی که برای ذخیره‌سازی آدرس‌هاشان داده شده پشته نام دارد. (LIFO)

stack

Stack register

اول آخرین آدرسی که گفته آمده برداشته می‌شود.

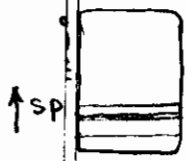


هر CALL آدرس برگشت را به بالای پشته می‌گذارد. RET آدرس را از بالای پشته برمی‌دارد.

در این حالت CALL یک آدرس دارد (که ضمنی است) و RET هم نیازی به آدرس ندارد.

اشاره‌گر stack یا stack pointer همیشه آدرس بالای پشته را در خود دارد.

در حالتی که آدرس‌ها زیاد است نمی‌توان تعداد زیاد رجیستر داشت و لذا دیگر از پشته رجیستر استفاده نمی‌کنیم بلکه از خود حافظه استفاده می‌کنیم که در این حالت مفهوم پشتهٔ حافظه مطرح می‌شود.



رشد stack در حافظه همیشه از آدرس‌های زیاد به آدرس‌های کم است.

پشته حافظه

CALL

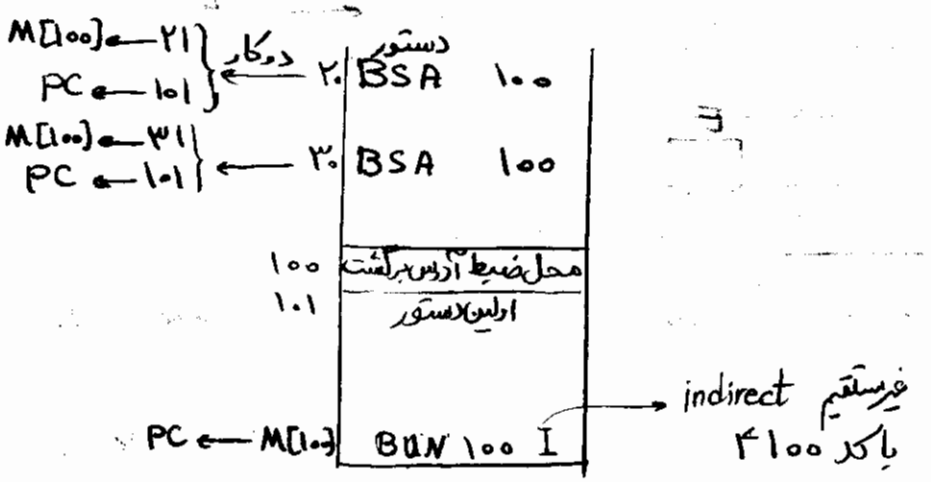
$$\left.\begin{array}{l} SP \leftarrow SP-1 \\ M[SP] \leftarrow PC \\ PC \leftarrow EA \end{array}\right\} \text{ضبط}$$

RET

$$\left.\begin{array}{l} PC \leftarrow M[SP] \\ SP \leftarrow SP+1 \end{array}\right\} *$$

		حافظه انتخاب کتاب
CALL	:	M[EA] ← PC PC ← EA + 1
RET		PC ← M[adr]

یک CALL دو آدرس لازم داشت. در نوع رجیستری یکی RAR و دیگری EA بود. در نوع پشته حافظه یکی SP و دیگری EA بود و در انتخاب کتاب یکی EA و دیگری EA+1 است.

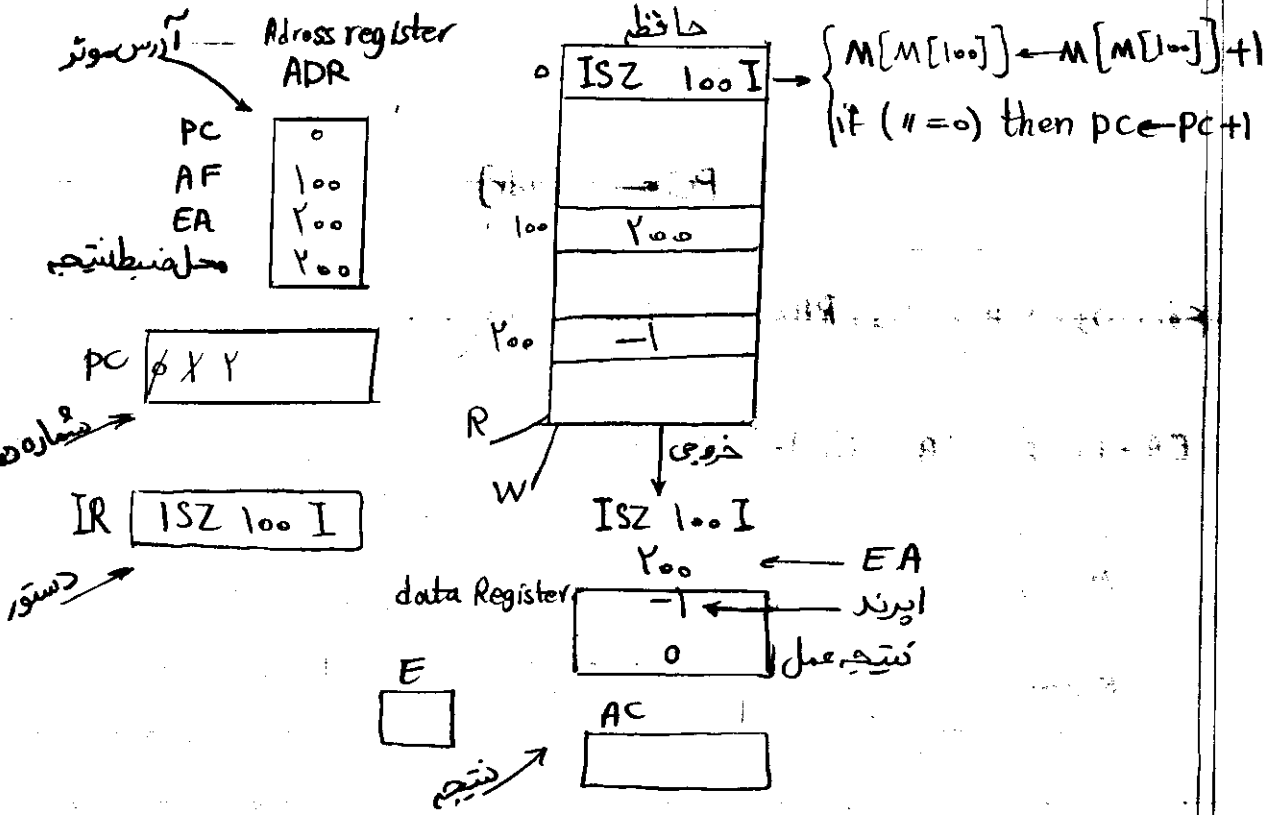


لذا در این حالت RET برخلاف دو حالت قبل آدرس صریح دارد. در این نوع خانهٔ اول هر زیربرنامه یک خانه برای آدرس گذاری است. همچنین برنامهٔ بازگشتی نمی‌توانیم داشته باشیم در صورتی که در پشته حافظه این امر امکان پذیر است.

آنچه که در جدول در زیر اثر دستورات نوشته شده همگی M-SP نیستند. مانند

M[EA] ← M[EA] + 1

پیاده سازی سخت افزار:



برای نگهداری اطلاعات خارج شده از حافظه یا ADR ها نیاز به رجیستر داریم. البته

Instruction Register

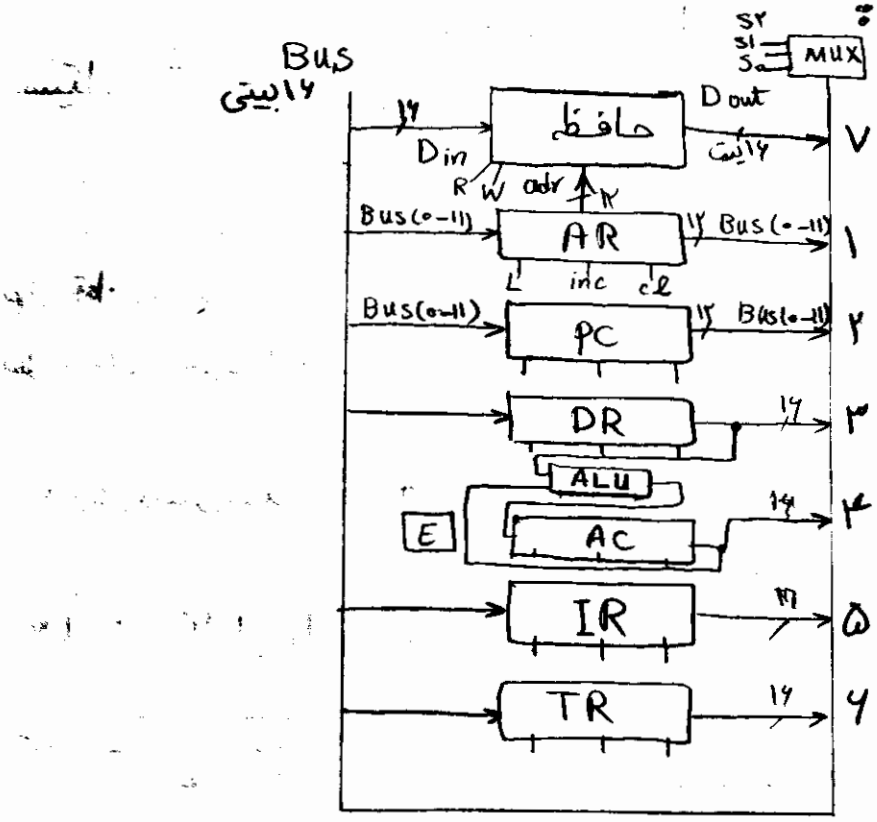
اگر در ادامه کار به آنها نیاز داشته باشیم. نام چنین رجیستری را IR می نامیم.

از AC نمی توانیم برای چنین عملی استفاده کنیم چون در اثر ISZ تغییر AC را نمی بینیم

یعنی AC نباید تغییر کند. ADR رجیستری است که برای نگهداری آدرس موثر به

کار می رود. تمام دستورات گفته شده با ترکیب سخت افزاری بالا اجرا خواهند شد

ساختار سخت افزاری :

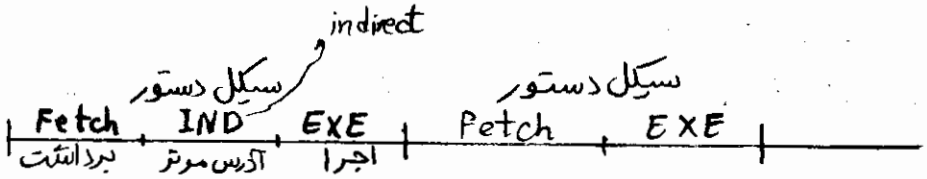


در مورد ارتباطات از Bus استفاده می کنیم. TR رجیستری است یدکی در صورت
Temporary register

تعریف دستور جدید در تمرینات از آن استفاده خواهیم کرد.

MUX نشان داده شده در بالا به عنوان سمبل Bus است و وجود خارجی ندارد.

محور زمان سیکل دستور است. در حین اجرای دستور می توان زیر سیکل هایی برای آن تعریف کرد.



اگر دستور حافظه ای و غیر مستقیم بود باید آدرس موثر را پیدا کنیم. عمل AC ← AC + M[EA] در زیر سیکل اجرا ، اجرا می شود.

سیکل دستور ← Fetch ۱
IND ۱
EXE ۲۵
INT (وقفه interupt) ۱
+ ← ۲۸ سیکل ماشین

برای هر دستور

برای تمام دستورها زیرسیکل‌های Fetch و IND یکسان است. ولی مرحلهٔ اجرا

مختلف

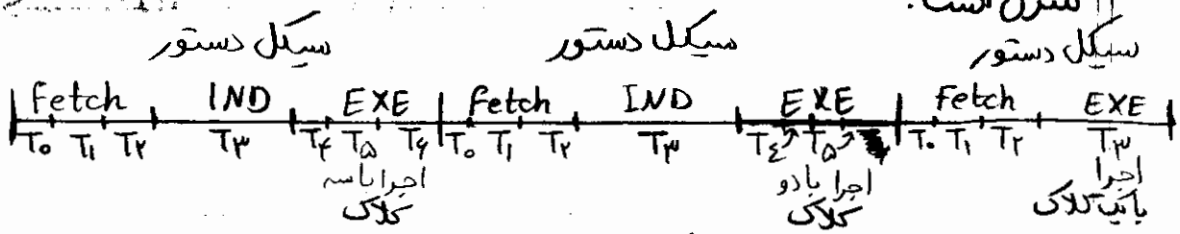
برای هر دستور به یک شکل است و چون ۲۵ دستور داریم ۲۵ سیکل اجرای داریم.

INT جزو سیکلهای ماشین است ولی جزو زیرسیکل‌ها نیست.

سیکل T جزو سیکل‌های ماشین است. مثلاً برای اجرای AC ← AC + M[EA] نیاز

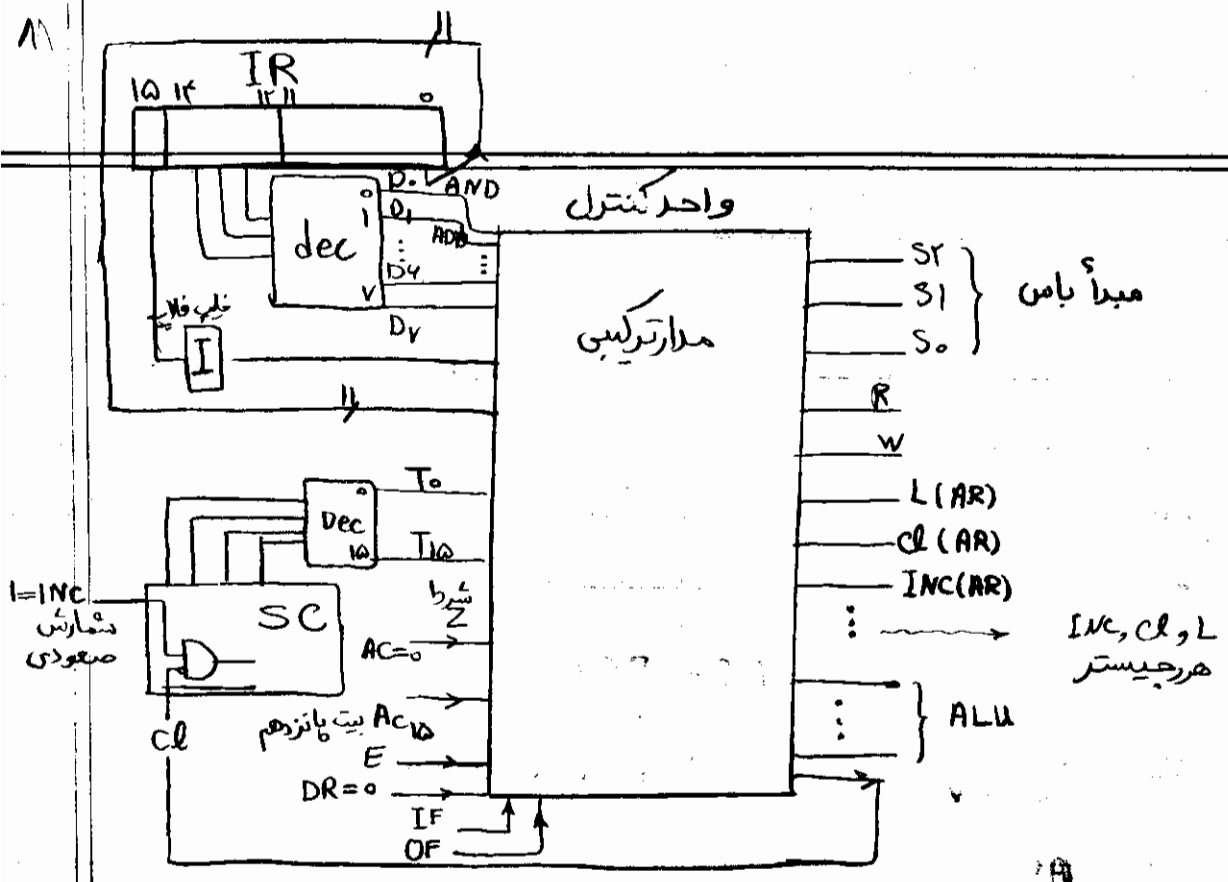
به سه کلاک داریم که از سیکل T استفاده می‌کنیم. هر سیکل T یک حالت از حالتهای واحد

کنترل است.



در درس میکروپروسسور سیکل‌های ماشین به نحو دیگری تعریف می‌شوند که شامل

INT ، IOW ، IOR ، MW ، MR ، Fetch است.



T_0 تا T_{15} توسط dec و sc تولید می‌شود و هر دستوری که حداکثر تا ۱۶ کلاک لازم داشته باشد می‌تواند اجرا شود. خواهیم دید که فقط به ۶ کلاک از اینها نیاز خواهیم داشت.

مقادیر اولیه در لحظهٔ روشن شدن کامپیوتر:

$PC = 0 \quad , \quad SC = 0$

Cl برای sc لازم است تا بعد از T_4 دوباره T_0 بشود. همچنین بر INC آن نیز غالب است که در شکل نمایش داده شده است.

بعد از مرحلهٔ Fetch نیاز داریم که بدانیم چه دستوری را آورده ایم و باید به IR مراجعه کنیم. ۱۱ بیت سمت راست IR طبق قرارداد کد شده است و فقط یک 1 دارد. لذا مستقیماً وارد واحد کنترل می‌شود.

از موارد دیگر موردنیاز واحد کنترل شرط‌ها هستند که فقط در دستورهای skip وجود داشتند.

اکنون مدار ترکیبی داخل واحد کنترل و ALU مانده است که باید معلوم شود.

نوشتن RTL :

سیکل Fetch

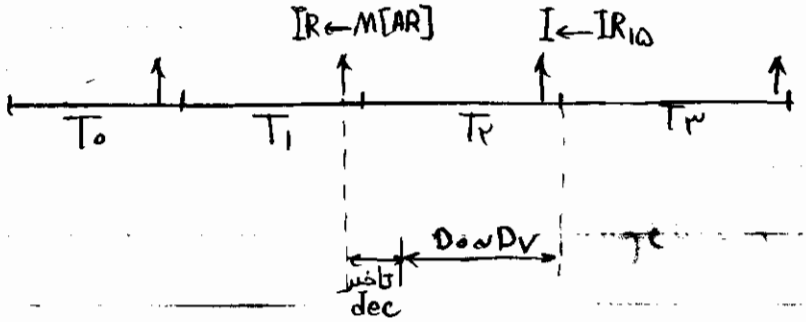
$T_0 : AR \leftarrow PC$

$T_1 : IR \leftarrow M[AR],\ PC \leftarrow PC+1$

$T_2 : D_0 \sim D_7 \leftarrow decode\,[IR(12-14)],\ I \leftarrow IR_{15}$

$AR \leftarrow IR(0-11)$

dec یک مدار ترکیبی است و نیاز به کلاک ندارد آنچه که در بالا آمده است انجام decode را صرفاً نشان می‌دهد. بعد از T_1، IR مقدار مطلوب خود را دارد و بعد از اندکی تأخیر dec، D_0 تا D_7 مقدار خود را گرفته اند ولذا در کلاک T_2 قابل استفاده خواهند بود. ولی ما در T_3 از آنها استفاده می‌کنیم ولذا کلاک T_2 اضافه است.



طلیپ‌فلاپ I نیز اضافه است. تنها چیزی که در کلاک T_2 مفید است، $AR \leftarrow IR(0-11)$ است که زودتر AR را آماده کرده است. اگر دستور حافظه‌ای بود این کار مفید است

واگر غیر حافظه ای بود هیچ ضرری نمی رساند.

نکات: $PC \leftarrow PC+1$ می تواند در T_0 قرار بگیرد.

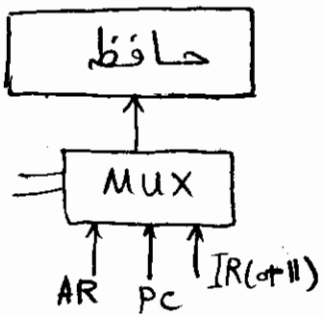
تمام سیکل گفته شده برای Fetch را می توان در دو کلاک داشت:

T_0 : $AR \leftarrow PC$, $PC \leftarrow PC+1$

T_1 : $Bus \leftarrow M[AR]$, $IR \leftarrow Bus$ AM
$I \leftarrow Bus_{15}$, $AR \leftarrow Bus(0-11)$

با این مدار نمی توان سیکل Fetch را در یک کلاک انجام داد. ولی اگر از PC مستقیماً به عنوان آدرس استفاده کنیم می توانیم در یک کلاک سیکل Fetch را انجام دهیم.

T_0 : $BUS \leftarrow M[PC]$, $IR \leftarrow Bus$
$AR \leftarrow Bus(0-11)$, $I \leftarrow Bus_{15}$, $PC \leftarrow PC+1$



مدار جدید برای انجام در یک پریود کلاک

پایان سیکل Fetch .

با نوشته شدن RTL Fetch حال می توانیم یک جمله از خروجی های زیر را داشته باشیم:

$L(AR) = T_0 + T_2$
$S_2 = 0 + T_1 + T_2$
$S_1 = T_0 + T_1 + 0$
$S_0 = 0$

$L(IR) = T_1$
$R = T_1$
$inc(PC) = T_1$
$I =$

حال T_0 و T_1 و T_2 گذشت و وارد T_3 شدیم. اکنون ۲۵ حالت وجود دارد که با یدیکی

انتخاب شود. اگر برفرض دستور CLA باشد: سیگنال اجرای دستور CLA EXE

CLA $\boxed{T_3 I' D_7 IR_{11} : AC \leftarrow 0 , SC \leftarrow 0}$

لذا زمان بعدی T_0 خواهد بود. حال دوباره سیکل fetch را طی می‌کنیم با این تفاوت

که این بار با $PC=1$ وارد سیکل Fetch می‌شویم. حال دستور CMA را داریم و....

V_{800}	CLA	$T_3 I' D_7 IR_{11} : AC \leftarrow 0 , SC \leftarrow 0$
V_{400}	CMA	$T_3 I' D_7 IR_{10} : AC \leftarrow \overline{AC} , SC \leftarrow 0$
〃	CLE	$T_3 I' D_7 IR_9 : E \leftarrow 0 , SC \leftarrow 0$
〃	CME ⟨INC	〃 $IR_8 : E \leftarrow E' , SC \leftarrow 0$
〃	⟨cir	〃 $IR_7 : AC \leftarrow AC+1 , SC \leftarrow 0$
〃	⟨cil	〃 $IR_6 : Cir\ E , AC$
	⟨SPA	〃 $IR_5 : cil\ E , AC$
	⟨SNA	〃 $IR_4 :$ if $AC>0$ then $PC \leftarrow PC+1 , SC \leftarrow 0$
	⟨SZA	〃 $IR_3 :$ if $AC<0$ 〃
	⟨SZE	〃 $IR_2 :$ if $AC=0$ 〃
	⟨HLT	〃 $IR_1 :$ if $E=0$ 〃
〃		〃 $IR_0 :$ توقف

حال سری دیگری از دستورات را خواهیم داشت که با یک کلاک T_3 انجام نمی‌شوند و

نیاز به کلاک بیشتر دارند.

AND مستقیم: $AC \leftarrow AC \wedge M[AF]$

AND مستقیم

$I'T_3D_0$: $DR \leftarrow \underbrace{M[AR]}_{\text{اپرند}}$

$I'T_4D_0$: $AC \leftarrow DR \wedge AC$, $SC \leftarrow 0$

AND غیرمستقیم: $AC \leftarrow AC \wedge M[M[AF]]$

AND غیرمستقیم

IT_3D_0 : $AR \leftarrow \underbrace{M[AR]}_{EA}$

IT_4D_0 : $DR \leftarrow \underbrace{M[AR]}_{\text{اپرند}}$

IT_5D_0 : $AC \leftarrow DR \wedge AC$, $SC \leftarrow 0$

همین گونه می‌توانیم برای بقیه دستورات بنویسیم. مشاهده می‌کنیم (در بالا) که مثلا

AC load برابر $I'T_4D_0 + IT_5D_0$ که ساده نمی‌شود. همین طور برای load

DR دو ترم با هم ساده نمی‌شوند. برای ساده‌سازی فرم زیر را در نظر می‌گیریم:

$I'T_3D_0$: ——

$I'T_4D_0$: $DR \leftarrow M[AR]$

$I'T_5D_0$: $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

→

سیکل اجرای AND چه مستقیم و چه غیرمستقیم

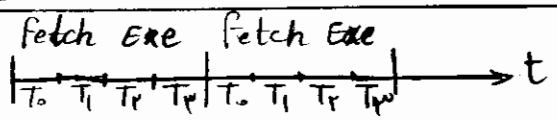
T_4D_0 : $DR \leftarrow M[AR]$

T_5D_0 : $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

سیکل indirect IND

$T_3I(D_0+D_1+\cdots+D_6) = D'_7$: $AR \leftarrow M[AR]$

حال مدار ساده‌تر می‌شود. فقط در حالت direct در T_3 کاری انجام نمی‌شود.



Fetch

T_0 : $AR \leftarrow PC$, $PC \leftarrow PC + 1$

T_1 : $IR \leftarrow M[AR]$

T_2 : $I \leftarrow IR_{15}$, $AR \leftarrow IR(0-11)$

IND

$T_3 D'_7 I$: $AR \leftarrow M[AR]$

EXE رجیستری

$T_3 D_7 I' IR_{11}$: $AC \leftarrow 0$, $SC \leftarrow 0$

// IR_{10} : $AC \leftarrow \overline{AC}$, $SC \leftarrow 0$

/ IR_9 : $E \leftarrow 0$, $SC \leftarrow 0$

⋮

$T_3 D_7 I' IR_1$: if E' then $PC \leftarrow PC+1$, $SC \leftarrow 0$

$T_3 D_7 I' IR_0$: توقف

AND

$T_4 D_0$: $DR \leftarrow M[AR]$

$T_5 D_0$: $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

} $AC \leftarrow AC \wedge M[EA]$

ADD

$T_4 D_1$: $DR \leftarrow M[AR]$

$T_5 D_1$: $E\,AC \leftarrow AC + DR$, $SC \leftarrow 0$

$EAC \leftarrow AC + M[EA]$

LDA

$T_4 D_2$: $DR \leftarrow M[AR]$

$T_5 D_2$: $AC \leftarrow DR$, $SC \leftarrow 0$

$AC \leftarrow M[EA]$

STA

$T_4 D_3$: $M[AR] \leftarrow AC$, $SC \leftarrow 0$

$M[EA] \leftarrow AC$

BUN

$T_4 D_4$: $PC \leftarrow AR$, $SC \leftarrow 0$

$PC \leftarrow EA$

BSA

$T_4 D_5$: $M[AR] \leftarrow PC, AR \leftarrow AR+1$

$T_5 D_4$: $PC \leftarrow AR, SC \leftarrow 0$

$M[EA] \leftarrow PC$
$PC \leftarrow EA+1$

ISZ

$T_4 D_4$: $DR \leftarrow M[AR]$

$T_5 D_4$: $DR \leftarrow DR+1$

$T_4 D_4$: $M[AR] \leftarrow DR, SC \leftarrow 0$
if $DR=0$ then $PC \leftarrow PC+1$

$M[EA] \leftarrow M[EA]+1$
if $M[EA]=0$ then $PC \leftarrow PC+1$

تمرین: دستور ISZ را انجام دهید و Inc DR , NOR DR (تست کردن) را با استفاده از AC انجام دهید. با توجه به اینکه تنها دو انتقالی که در یک کلاک داریم انتقالهای زیری تواند باشد:

$AC \leftarrow DR$ (از طریق ALU), $DR \leftarrow AC$ (از طریق باس)

باید توجه کرد که دو انتقال از طریق باس در یک کلاک امکان پذیر نیست.

جواب:

$T_4 D_4$: $DR \leftarrow M[AR]$

$T_5 D_4$: $AC \leftarrow DR, DR \leftarrow AC$

$T_4 D_4$: $AC \leftarrow AC+1$

$T_7 D_7$ $M[AR] \leftarrow AC, AC \leftarrow DR, SC \leftarrow 0$
if $AC=0$ then $PC \leftarrow PC+1$

$$\underset{\text{clear}}{cl}(SC) = \underbrace{T_3 I' D_7 (IR_{11} + \cdots + IR_0)}_{\text{رجیستری}} + \underbrace{T_5 D_0 + T_5 D_1 + \cdots + T_4 D_4}_{\text{حافظه‌ای}} + \underbrace{T_3 I D_7 (IR_{11} + \cdots + IR_4)}_{\text{ورودی/خروجی}} \rightarrow T_3 D_7$$

بحث کدهای تعریف نشده :

کدهای مقابل در حافظه وجود دارند و کامپیوتر را روشن می‌کنیم. سوال : چه وضعیتی پیش می‌آید؟

۰	7000	NOP (No operation)
۱	7C00	
۲	7801	

دستور اول fetch می‌شود و کلاک T_0 و T_1 و T_2 می‌گذرند. در T_3 کاری انجام نمی‌شود و تا T_{15} هیچ کاری انجام نمی‌شود. بعد وارد T_0 می‌شود و دستور بعدی را fetch می‌کند و... . چنین دستوری NOP نام دارد و فقط PC تغییر می‌کند.

کاربرد : فرض کنیم چند خط دستور داریم و می‌خواهیم یکی از دستورها را حذف کنیم. برای اینکه شماره خطها عوض نشود به جای دستور حذف شده از NOP استفاده می‌کنیم.

```
  ⋮
LDA X
ADD Y
NOP
STA U
  ⋮
```

سوال : آیا می‌توان به جای NOP ، ۰۰۰۰ گذاشت؟ خیر چون این کد ، کد دستور AND است و در آن کاری انجام خواهد شد.

دستور VA00 باعث می‌شود Cl(Ac) ، Cl(E) فعال شوند.

دستور VCOO چون باعث می‌شود هم cl(AC) فعال شود و هم $\overline{AC} \leftarrow AC$. درنتیجه برای ما معلوم نخواهد بود کجا می‌شود و این دستور به درد نمی‌خورد.

دستور ۷۸۰۱ باعث می‌شود که cl (AC) فعال شود و سپس دستور توقف اجرا شود.

برای اینکه در چنین دستورهایی که دوکار انجام می‌شود فقط یکی انجام شود مثلاً فقط cl(AC) انجام شود در فرمان آن که $T_3 D_7 I' IR_{11}$ است جمله‌های زیر اضافه می‌شود $(\dot{IR}_{10} \cdots \dot{IR}_0)' T_3 D_7 I'$. اگر برای تمام دستورها چنین کاری انجام دهیم تمام کدهای تعریف نشده ، NOP خواهند بود.

تمرین : مدادی طرح کنید که در چنین کامپیوتری اگر سگنال (دستور) غیرمعتبر fetch شد، یک سگنال مبنی بر غیرمعتبر بودن دستور تولید شود.

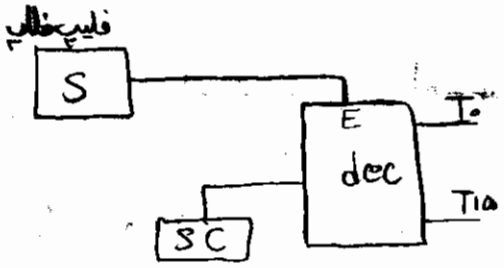
توقف: وقتی دستور HLT در T_2 تا T_0 ، fetch می‌شود در سیکل اجرای آ $1 \leftarrow pc - pc$ را قرار می‌دهیم که باعث توقف خواهد شد. این امر مستلزم داشتن dec برای PC است.

روش دیگر این است که زمان همیشه در T_3 باقی بماند. برای این منظور در سیکل اجرا $3 \leftarrow SC$ را قرار می‌دهیم. این امر مستلزم آن است که SC ، یک ورودی دارای کد ۳ و یک load

داشته باشد که این load بر cl و inc آن برتری داشته باشد.

روش سوم این است که تمام زمانها را صفر کنیم و به این مفهوم که در ادامه هیچ زمانی نداریم.

برای این منظور از Enable دکودر استفاده می‌کنیم.



در سیکل اجرای HLT خواهیم داشت : $S \leftarrow 0$

سوال:
برای اینکه بعد از توقف دوباره کارها انجام شود و از HLT خارج شویم دستور STR را تعریف می‌کنیم : $D_7T_3IR_0 : \begin{matrix} S \leftarrow 1 \\ SC \leftarrow 0 \end{matrix}$ — STR یا کد F001

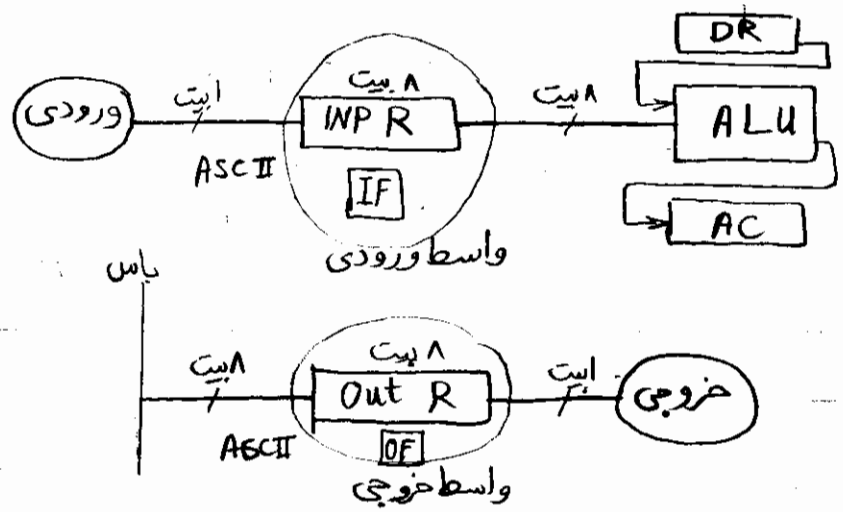
آیا این دستور ما را از HLT خارج می‌کند؟ خیر. فرم دستور درست است امّا چون بعد از توقف زمان صفر است این دستور Fetch و اجرا نمی‌شود. لذا هرگونه خارج شدن از HLT نرم افزاری نمی‌تواند باشد و باید از طریق سخت افزار انجام شود.

سیگنال Reset که به این منظور ارسال می‌شود PC و SC را صفر می‌کند و S هم 1 می‌شود.

تکلیف ۳: فصل ۵ : ۱۲، ۱۳، ۱۴، ۱۵، ۱۶، ۱۷، ۱۸، ۱۹، ۲۳

بحث ورودی و خروجی:

دستورات I/O	کد	اثر دستور
INP	F۸۰۰	AC(۰-V) ← INPR , IF ← ۰
OUT	F۴۰۰	outR ← AC(۰-V) , OF ← ۰
SKI	F۲۰۰	if IF then pC ← PC+۱
SKO	F۱۰۰	if OF then PC ← PC+۱
ION	F۰۸۰	IEN ← ۱
IOF	F۰۴۰	IEN ← ۰



INP

$T_۳ I C_V IR_{۱۱}$: AC(۰-V) ← INPR, IF ← ۰ , SC ← ۰

OUT

$T_۳ I D_V IR_{۱۰}$: outR(۰-V) ← AC(۰-V) , OF ← ۰ , SC ← ۰

SKI

$T_۳ I D_V IR_۹$: if IF then PC ← PC+۱ , SC ← ۰

SKO

$T_۳ I C_V IR_۸$: if OF then PC ← PC+۱ , SC ← ۰

ION

$T_۳ I C_V IR_۷$: IEN ← ۱ , SC ← ۰

IOF

$T_۳ I D_V IR_۶$: IEN ← ۰ , SC ← ۰

درمیان کدهای نامعتبر فقط یکی با ۷ شروع می‌شود. (۷۰۰۰)

F $IR_{11} \cdots IR_6 | IR_5 \cdots IR_0$

اگر $IR_6 \cdots I_{R_{11}}$ برابر صفر باشند نامعتبر است.

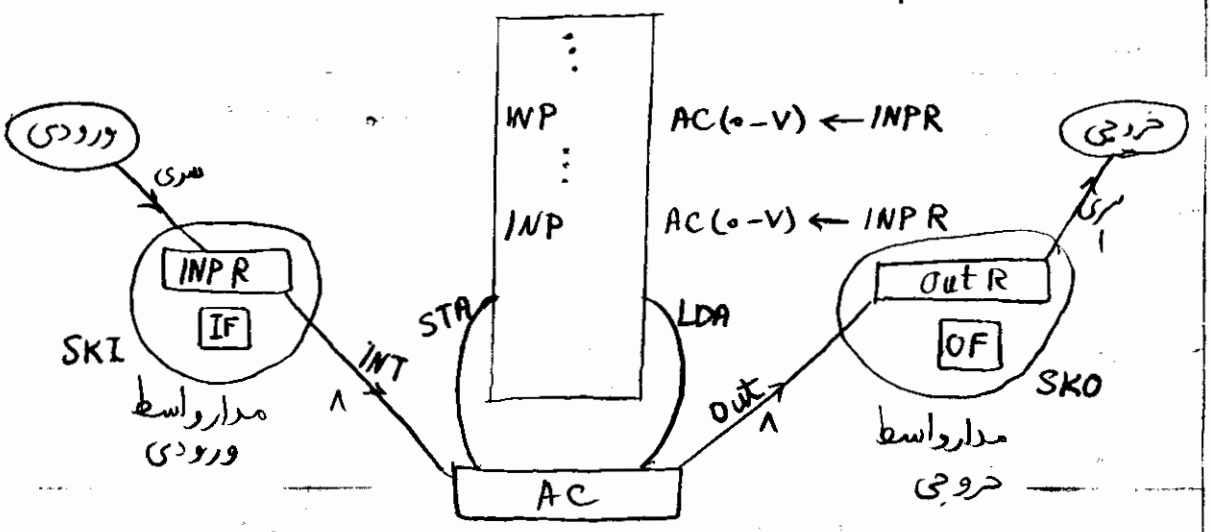
تعداد کل NOP $= 2^6$ ←

مشکلات / روش‌ها	کندی دستگاه: تست آماده بودن ورودی	تست آماده بودن خروجی	خواندن ورودی	نوشتن خروجی	انتقال به حافظه	برداشت از حافظه	پردازش	پردازش
programmed I/O	SKI در CPU به کمک مدار واسط	SKO در CPU مدار واسط	INP در CPU به کمک مدار واسط	out در CPU	STA در CPU	LDA	ADD AND ⋮	ADD AND
INT driven I/O								
DMA								
I/O processor	مدار واسط	مدار واسط	مدار واسط	مدار واسط		مدار واسط		مدار واسط

ادامهٔ مشکلات: کداستفاده شده وطول کد وطرزارسال

مدار واسط انجام می دهد.

مدار واسط



بعد از خواندن ورودی، دستور به فرورودی در حافظه انتقال یافته و در آن ذخیره

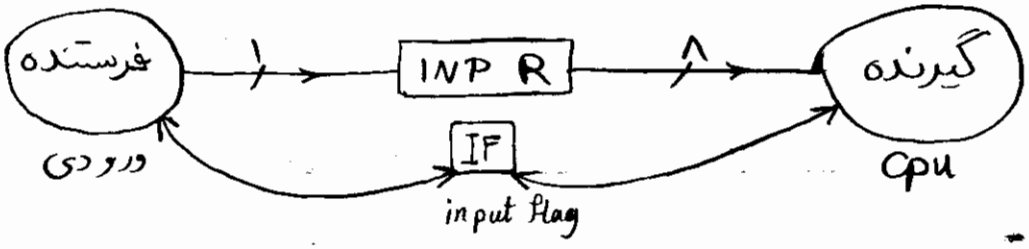
می‌شود. بعد روی آن ها پردازش انجام می‌شود که شامل تبدیل به Binary و ... است.

در مورد دستورات خروجی مراحل عکس بالا طی می‌شود (در این مورد با فرخروجی داریم).

تبدیل کد سری به موازی در مورد ورودی های غیر ASCII است که کد آنها باید به ASCII تبدیل شود.

	چک آماده بودن ورودی خروجی	خواندن نوشتن	انتقال به حافظه / بافر ورودی	برداشت از حافظه / بافر خروجی	پیش پردازش / پردازش بعدی	تبدیل کد سری به موازی
	cpu و مدار واسط	cpu و مدار واسط	cpu	cpu	cpu	cpu و مدار واسط
Prog I/O	SKI SKO	INP OUT	STA LDA		ADD، AND، ...	
INT I/O	مدار واسط	//	//	//	//	//
DMA	مدار واسط	مدار واسط	مدار واسط	مدار واسط	//	مدار واسط
I/O Proces.	//	//	//	//	مدار واسط	//

از بالا به پائین بار کاری cpu کم شده و سخت افزار افزایش می یابد.



فرستنده اطلاعات را می فرستد و IF را یک می کند IF←1

گیرنده اطلاعات را برمی دارد و IF را صفر می کند. IF←0

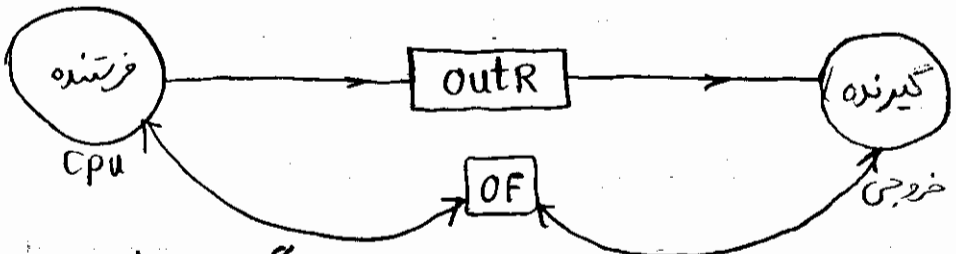
فرستنده اگر IF=0 باشد می‌فرستد و گیرنده اگر IF=1 باشد بر می‌دارد.

```
INP     AC(0-7) ← INP R , IF ← 0
SKI     if IF=1 then PC ← PC+1
```
(IF روی skip)

```
10  SKI
11  BUN 10
12  INP
    STA IB
```

سیکل خواندن در بالا نشان داده شده است. مشاهده می‌کنیم که بیشترین تأخیر در Cpu

هنگام تست آماده بودن است و در این مدت وقت cpu تلف می‌شود.



اگر OF=1 است می‌فرستد و OF ← 0

اگر OF=0 است بر می‌دارد و OF ← 1

```
SKO     if OF=1 then PC ← PC+1
OUT     OutR ← AC(0-7) , OF ← 0
```

```
1  SKO
2  BUN 1
3  LDA OB
4  OUT
```

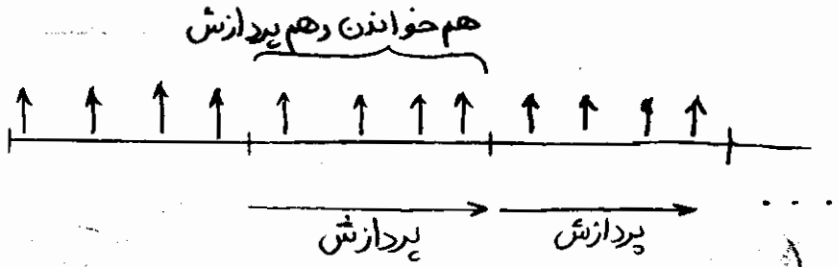
LDA OB = load AC from output Buffer

STA IB = store AC in input Buffer

روش بالا programmed I/O بود. در روش INT I/O حلقه‌های انتظار حذف

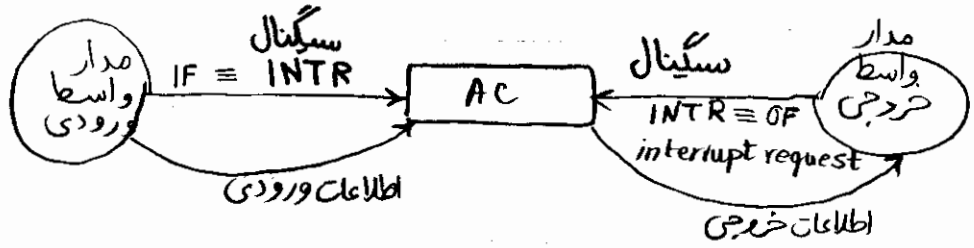
می‌شود و در مدت انتظار cpu کار مفید انجام می‌دهد. لذا چک آماده بودن را مدار واسط انجام می‌دهد و به cpu اعلام می‌کند.

مسئله: می‌خواهیم حین پردازش ورودی را نیز بخوانیم:



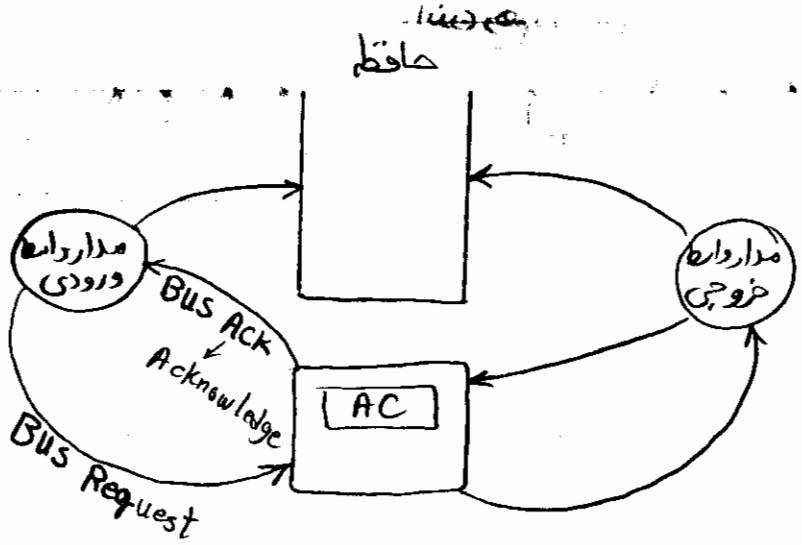
اگر زمان بین ورودی‌ها ثابت باشد مسئله بالا با روش prog. I/O مکافات است و اگر این زمان‌ها نابرابر باشند غیرممکن است.

در روش INT I/O (وقفه) در حین پردازش اگر ورودی باشد توسط مدار واسط اعلام می‌شود و در این لحظه پردازش متوقف می‌شود و ورودی خوانده می‌شود.



در روش DMA مدار واسط ورودی مستقیماً به حافظه دسترسی دارد و وقتی ورودی آماده شد مستقیماً آن را به حافظه انتقال می‌دهد. تنها کاری که انجام نمی‌شود پردازش

اطلاعات ورودی است. با توجه به اینکه ورودی وخروجی به حافظه توسط یک Bus انجام می‌شد لذا در این روش همزمان مدار واسط خروجی و مدار واسط ورودی نمی توانند به حافظه اطلاعات دهند لذا سیگنال هایی باید بین این مدارها و حافظه رد و بدل شود.

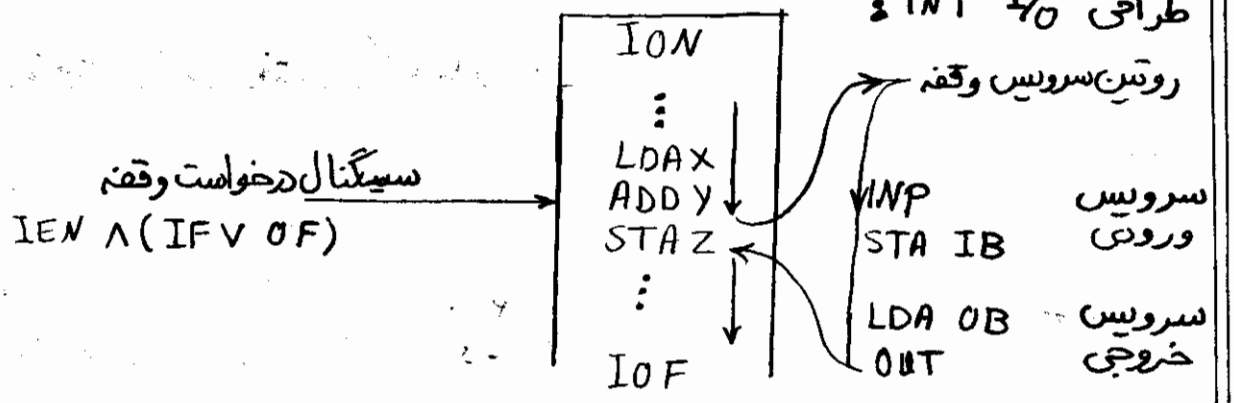


در این روش فقط یک پریود کلاک نیاز است تا C.PU ، Bus را خالی کند. مدار واسط ورودی Bus Req را می فرستد و پس از دریافت Bus Ack ، اطلاعات را در حافظه قرار می دهد.

در روش I/O processor تمام مراحل مانند روش DMA است ولی در مدارهای واسط خروجی و ورودی پردازش نیز وجود دارد. به عنوان مثال یکی از پردازش هایی که در این مدارها انجام می‌شود تبدیل ASCII به Binary است.

در روش prog. I/O زمان اجرای ۵۰۰۰۰ دستور تلف می‌شود و در INT I/O زمان اجرای ۲۰ دستور تلف می‌شود و در DMA، I/O process. زمان یک کلاک تلف می‌شود.

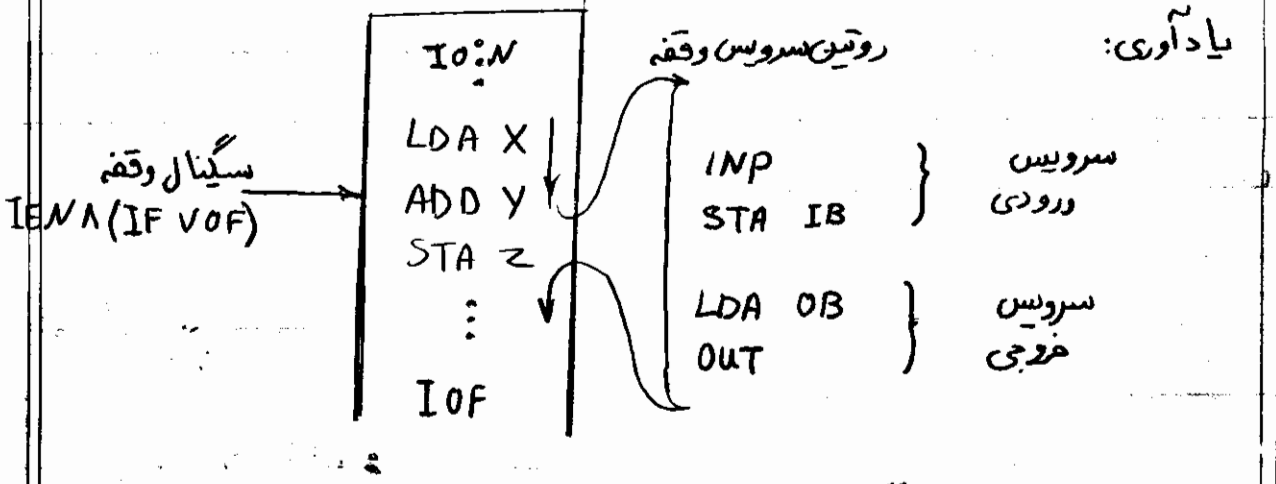
طراحی INT I/O:



بسته به اهمیت ورودی ها که تعریف می‌شود اولویت صدور وقفه برای ورودی یا خروجی و یا ادامه کارها مشخص می‌شود. برای روشن شدن این اولویت از یک بیت IEN (interrupt enable) استفاده می‌شود که در ابتدا مقدار اولیهٔ آن صفر است. فعال شدن این بیت با دستور ION است. صفرشدن آن به سه شکل است: یکی مقدار اولیه آن که صفر است. یکی دستور IOF (در سیکل وقفه) است و دیگر را خواهیم دید.

لذا باید سیگنال درخواست وقفه و سیکل وقفه و روتین سرویس وقفه مشخص شوند.

یادآوری:



وضعیت برنامه در رفت و برگشت ها (برای وقفه) باید حفظ شود.

اطلاعات وضعیت برنامه شامل موارد زیر است که باید حفظ شوند:

۱- PC که ترتیب اجرا را نشان می دهد - ۲- AC و E که پردازش ها در آنها انجام گرفته و مقدار دارند. ۳- حافظه: که برنامه نویس با تخصیص مناسب فضای آن می تواند این مشکل را حل کند.

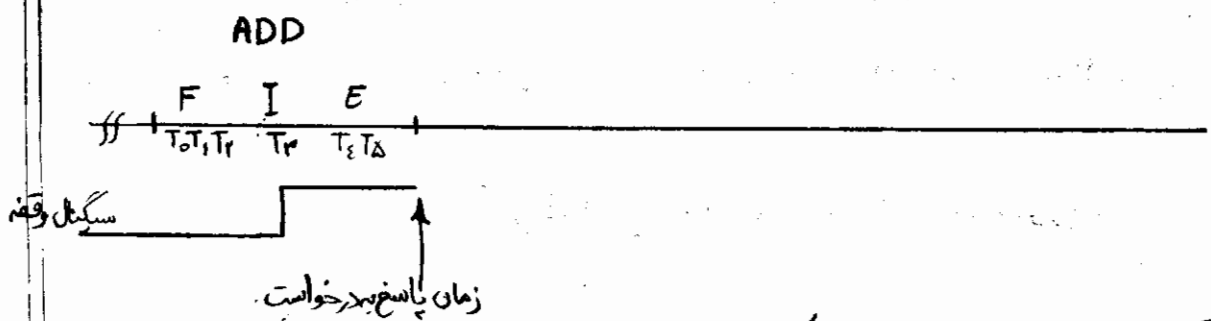
درستجی PC، AC، E باید در سرویس وقفه ضبط شوند:

روتین سرویس وقفه

```
STA  TA   } ضبط E,AC
Cir       }
STA  TE   }
          } سرویس ورودی
          } سرویس خروجی
LDA  TE   }
cil       } بازیابی AC, E
LDA  TA   }
```

سوال: آیا حفظ مقادیر IR , AR , DR , SC , S مهم نیست؟

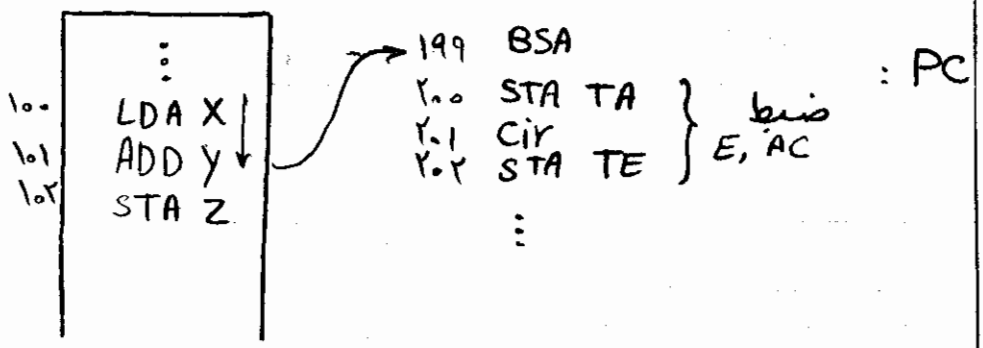
حفظ اینها بستگی به زمان پاسخ به وقفه دارد:



سیگنال وقفه در هر پریودی از کلاک می تواند فعال شود. امّا زمان رها کردن برنامهٔ اصلی و پاسخ به درخواست در آخر سیکل یک دستور است و دستور را نصفه کاره نمی گذاریم. اگر بخواهیم چنین عمل کنیم (یعنی دستور را نیمه کاره رها کنیم) باید تمام رجیسترهای بالا را ضبط کنیم.

رجیسترهای بالا، رجیسترهای کمکی هستند که فقط در طول زمان اجرای دستور به آنها نیاز مندیم.

پس حداکثر تأخیری که برای پاسخ به وقفه وجود دارد ۴ پریود کلاک است.



دستور BSA برای ضبط PC موجود است امّا قابل استفاده نیست. هیچ راه نرم افزاری برای ضبط PC وجود ندارد و باید سخت افزاری آن را ضبط کرد.

سوال امتحانی: دستوری برای ضبط PC طرح کنید:

BSA : $M[EA] \leftarrow PC$, $PC \leftarrow EA + 1$

store
STPC F001 $ID_7 T_3 IR_0$: $TR \leftarrow PC$, $SC \leftarrow 0$

Restore
RPC F002 $ID_7 T_3 IR_1$: $PC \leftarrow TR$, $SC \leftarrow 0$

می‌توانیم دستورات بالا را طرح کنیم امّا هیچ کدام مفید نخواهد بود چون:

در طرح صفحهٔ قبل برای اینکه از خط ۱۰۱ به خط ۱۹۹ برویم اگر بخواهیم PC=۱۰۲ را حفظ کنیم PC، ۱۹۹ نشده و به خط ۱۹۹ نمی‌رویم و اگر PC، ۱۹۹ شده و وارد سیکل وقفه بشویم PC=۱۰۲ را از دست می‌دهیم.

لذا در وقفه حداقل دو کار باید انجام شود: ۱- ضبط PC که در اینجا ۱۰۱ است.

۲- مقداردهی به PC که در اینجا ۱۹۹ است. این دو کار فقط سخت‌افزاری انجام می‌شود.

سیکل وقفه { ضبط PC ; مقداردهی به PC }

برای رفتن به روتین وقفه باید سیکل وقفه انجام شود.

محل ضبط PC (۳ انتخاب) ← رجیستر
← پشته حافظه
← قبل از اولین دستور روتین وقفه

از انتخاب سوم استفاده می‌کنیم. به خاطر اینکه با CALL نگیو اختی باشد و دستور جدیدی اضافه نکنیم. (برای ضبط در پشته و یا رجیستر دستور داریم ولی برای بازیابی آن باید دستور تعریف کنیم).

آدرس برگشت ۱۹۹

ضبط E, AC ۲۰۰ ۲۰۱ ۲۰۲

⋮

انتخاب سوم :

BUN 199 I : PC ← M[199] بازیابی PC

مقداردهی به PC

- PC ← Const ✓
- PC ← M[Const]
- PC ← BUS

مورد اول را انتخاب می‌کنیم. (موارد بعدی هم می‌توانند انتخاب شوند)

Const=1 را قرار می‌دهیم.

سیکل وقفه (سیکل بیست و هشتم)

$$\begin{cases} M[0] \leftarrow PC \\ PC \leftarrow 1 \end{cases}$$

سیکل وقفه

T_8 : AR ← 0

T_9 : M[AR] ← PC , AR ← AR+1

T_{10} : PC ← AR

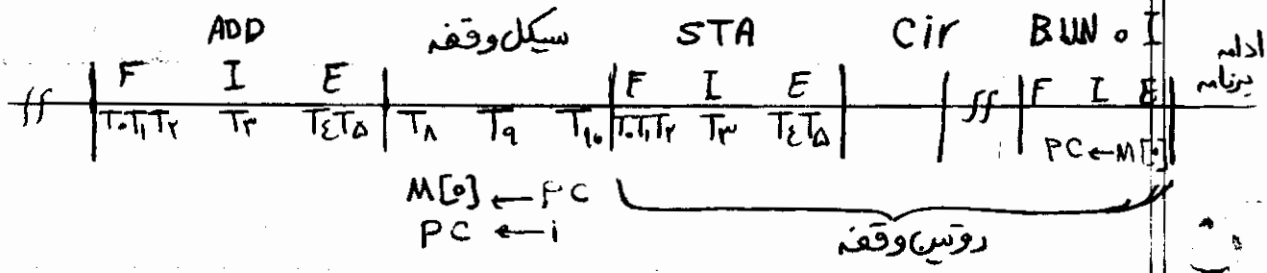
یا

سیکل وقفه

AR ← 0

M[AR] ← PC, PC ← 0

PC ← PC+1



لذا در سیکل دستور بعد از T_5 یا T_6 داریم یا T_8 . در قبل در آخرین کلاک دستور

داشتیم : $T_{(F)}$ Final : $SC \leftarrow 0$

اکنون آن را اینگونه تغییر می دهیم :

T_F : if $[IEN \wedge (IFVOF)]$ then $SC \leftarrow 8$ else $SC \leftarrow 0$

باید قابلیت load برای مدار SC اضافه شود.

سگنال وقفه ← سیکل وقفه ← روتین وقفه .

در مراحل بالا باید کارهای زیر انجام بگیرد :

۱ـ ضبط وضعیت ۲ـ تشخیص عامل وقفه ۳ـ اولویت بین عوامل

۴ـ رفتن به روتین سرویس عاملی که اولویت دارد . ۵ـ بازیابی وضعیت .

موارد ۱و۲و۳ را نرم افزاری انجام دادیم می توانند سخت افزاری انجام شوند . ۴ را سخت افزاری انجام دادیم و می تواند نرم افزاری انجام گیرد.

تکمیل بحث روتین وقفه :

روش ارائه شده روش polling است .

۰	آدرس برگشت	
۱	STA TA	ضبط وضعیت
۲	Cir	
۳	STA TE	
۴	SKI	تعیین عامل
۵	BUN ۱۰	
۶	INP	سرویس ورودی
۷	STA IB	
۸	SKO	تعیین عامل
۹	BUN ۱۲	
۱۰	LDA OB	سرویس خروجی
۱۱	Out	
۱۲	LDA TE	بازیابی
۱۳	cil	
۱۴	LDA TA	
۱۵	ION	
۱۶	BUN ۰I	

این روش مشکل دارد. بعد از STA چون سگنال وقفه وجود دارد باز هم وقفه رخ می‌دهد و همین طور تا بینهایت بعد از STA، وقفه، بعد از آن STA و ... انجام می‌شود. لذا باید به طریقی در روتین وقفه، عامل وقفه قطع شود. برای قطع عامل وقفه یا باید از IOF استفاده کرد و یا به صورت سخت افزاری IEN یا OFVLF را صفر کرد. ما IEN=۰ را انتخاب می‌کنیم. چون این کار باید انجام شود، لذا آن را سخت افزاری تعریف می‌کنیم و در سیکل وقفه قرار می‌دهیم. اگر از IEN=۰ استفاده نکنیم باید از IOF استفاده

کنیم. ION در آخر روتین وقفه برای فعال کردن دوباره عامل وقفه قرار می‌گیرد.

سیکل وقفه جدید

T_8 :

T_9 :

T_{10} : ——————>, IEN ← 0

حال باید تضمین کنیم بعد از ION (خط ۱۵) وقفه اتفاق نمی افتد.

T_3 : IEN ← I, if IEN(IFVOF) then SC ← 8 else SC ← 0

بعد از کلاک پر می‌شود — مقدار قبلی که صفر است

بعد از BUN در خط ۱۶ اگر وقفه اتفاق افتد اشکالی ندارد.

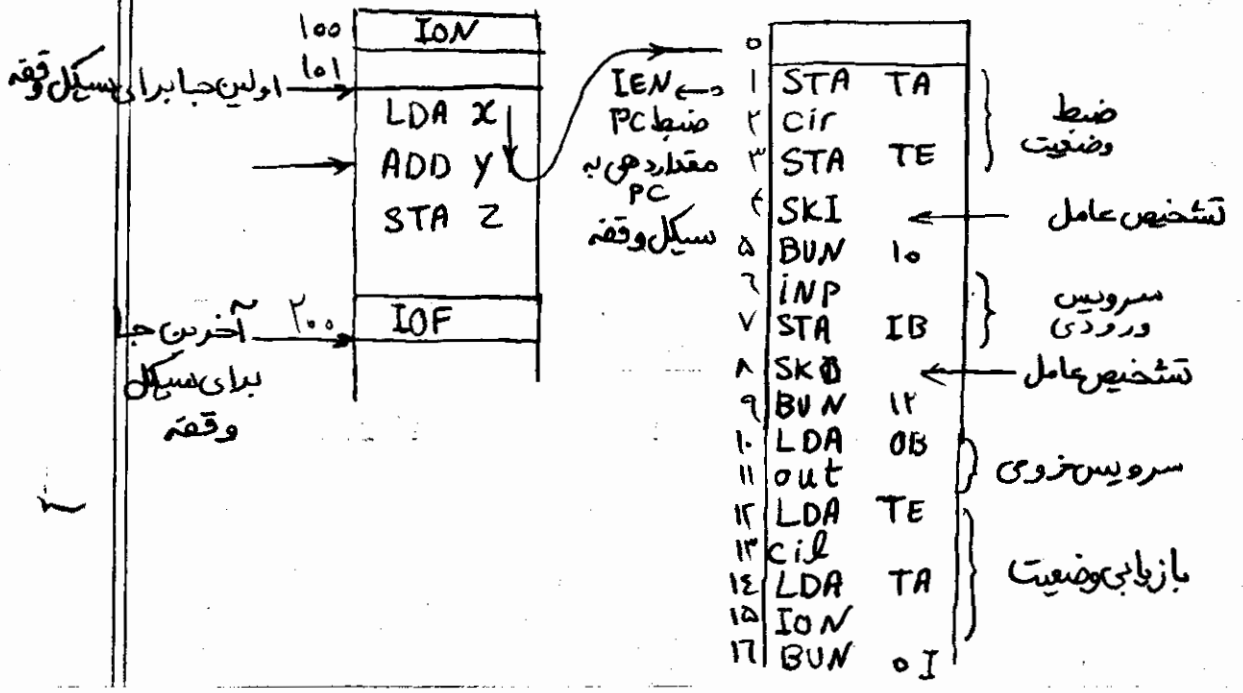
یادآوری :

T_8 : AR ← 0

T_9 : M[AR] ← PC, AR ← AR+1

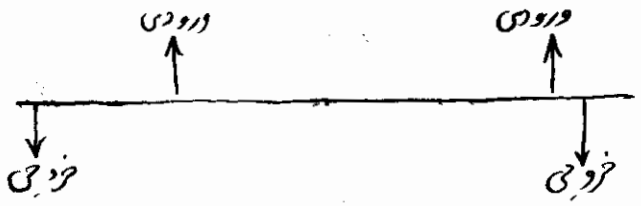
T_{10} : PC ← AR , IEN ← 0 , SC ← 0

T_F : if IEN(IFVOF) then SC ← 8 else SC ← 0



بعد از ION در خط ۱۰۰، وقفه بعد از ۱۰۱ می‌تواند اتفاق افتد. چون در قبل از آن مقدار قبلی چک می‌شود. به همین ترتیب اگر در خط ۲۰۰، IOF باشد آخرین جا مناسب است. برای سیکل وقفه بعد از ۲۰۰ خواهد بود که نامناسب است.

در فاصلهٔ LDATE تا BUN 0 I ، IF ، OF می‌توانند یک باشند که دو حالت دارد: یکی برای دستگاههای ورودی و خروجی سریع - دیگری برای دستگاههای کند به طوری که فاصلهٔ دو ورودی و دو خروجی از هم زیاد است امّا زمان بین یک ورودی و خروجی از هم کم است.



در مورد آخرین جا برای سیکل وقفه، آخرین جا بعد از IOF است که ما نمی‌خواهیم بعد از آن وقفه اتفاق افتد. لذا این دستور را استثناء می‌کنیم و سیکل وقفه به صورت زیر خواهد بود:

T_8 :

T_9 :

T_{10} :

T_3 : iF IEN (IF VOF) then (SC ← 1) else (SC ← 0)
, IEN ← 0

IOF T_3 : IEN ← 0 , SC ← 0

روش پیاده سازی کتاب :

$R \leftarrow 0$ مقداراولیه

$T_F: SC \leftarrow 0$

سیکل وقفه	سیکل Fetch
$R\,T_0: AR \leftarrow 0$	$R'\,T_0: AR \leftarrow PC$
$R\,T_1: M[AR] \leftarrow PC, AR \leftarrow AR+1$	$R'\,T_1: IR \leftarrow M[AR], PC \leftarrow PC+1$
$R\,T_2: PC \leftarrow AR, IEN \leftarrow 0, SC \leftarrow 0, R \leftarrow 0$	$R'\,T_2: AR \leftarrow IR(0\text{-}11), I \leftarrow IR_{15}$

برای تشخیص دوسیکل بالا یک فلیپ فلاپ R را تعریف می کنیم.

$IEN(IFVOF): R \leftarrow 1$

$(IOF)'(T_4+T_5+T_6+T_7)\,IEN(IFVOF): R \leftarrow 1$

$T_2' \cdot T_1' \cdot T_0' \cdot IEN(IFVOF): R \leftarrow 1$

درهزم بالا اگر وقفه بعین T_3 تا T_4 اتفاق بیفتد در T_0 بعدی رویت می شود.

اگر T_2 را نیز به زمانهای بالا (درداخل پرانتز) اضافه کنیم در زمان RT_2 ، $R \leftarrow 0$ ، $R \leftarrow 1$ با هم اتفاق می افتد. برای رفع این اشکال می توان $IEN \leftarrow 0$ را در RT_1 قرارداد.

تمرین : کامپیوتر فصل ۵ را طوری تغییر تا روتین های ورودی وخروجی بشکل مقابل باشد :

0	
1	INP
2	STA IB
3	ION
4	BUN 0 I
...	
8	
	LDA OB
	OUT
	ION
	BUN 1 I

روتین ورودی

روتین خروجی

آنچه که از فصل ۵ باقیمانده است مدارهای ترکیبی واحد کنترل و ALU است:

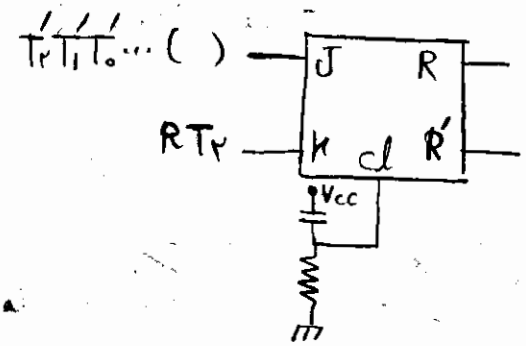
X_0 : AR ← IR
X_1 : M[AR] ← AC
X_2 : PC ← AR
X_3 : IR ← M[AR]

(اگر تمام دستورات معتبر باشند در یک لحظه نقط یک سطر RTL یک شده و انجام می‌شود).

$S_2 = X_0 + X_1 + 0 + X_3$
$S_1 = 0 + 0 + X_2 + X_3$
$S_0 = X_0 + 0 + 0 + X_3$
$R = X_3 + X_7$
$W = X_1 + \cdots$
$INC(PC) = R'T_0 + I'D_7T_3IR_1E' + D_7T_3IR_2AC_8 + IR_4Z$ ($E=0$, >0, $=0$)
$+ IR_3Z'AC_{15}$ (>0)
⋮

برای فلیپ فلاپ R:

$R'(IOF)'(\cdots + T_4 + T_3 + \cdots)IEN(IFVOF)$: $R \leftarrow 1$
حالت اولیه : $R \leftarrow 0$
RT_2 : $R \leftarrow 0$



در مورد AC :

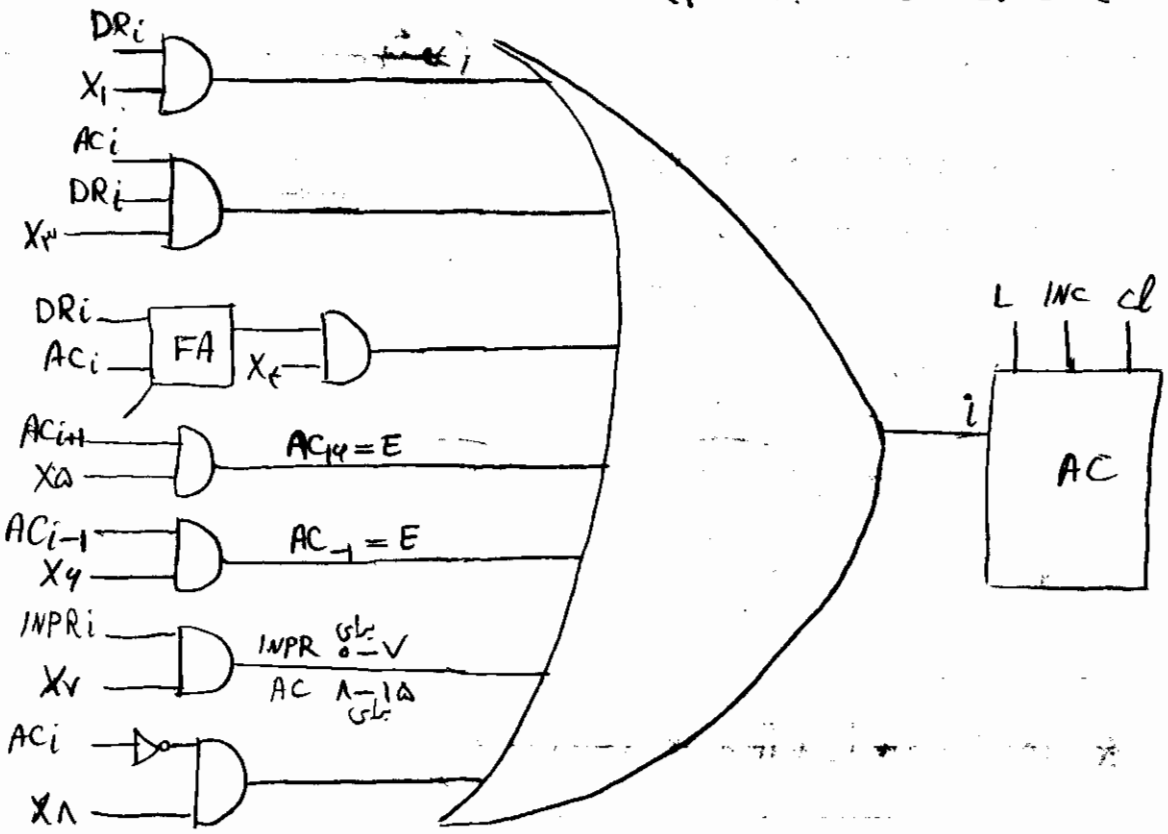
X_0 : AC ← 0
X_1 : AC ← AC + 1
X_2 : AC ← DR
X_3 : AC ← AC ∧ DR
X_4 : AC ← AC + DR
X_5 : E, AC ← cir E, AC
X_6 : E, AC ← cil E, AC
X_7 : AC(0-7) ← INPR
X_8 : AC ← $\overline{AC}$

$L(AC) = X_3 + X_4 + X_5 + X_6 + X_7 + X_8$

$cl(AC) = X_0$

$INC(AC) = X_1$

نمایش یک بیت از AC (بیت i ام)



فصل ۶ : نرم افزار سیستم

نرم افزارهای سیستم شامل :

ادیتور - مترجم - لینکر - لودر - دیباگر - توابع کتابخانه ای - سیستم عامل

سیستم عامل وظیفهٔ مدیریت و کنترل نرم افزاری منابع سیستم را به عهده دارد که

منابع سیستم شامل : ورودی/خروجی ، زمان CPU ، حافظهٔ اصلی ، حافظهٔ کمکی و برنامه ها

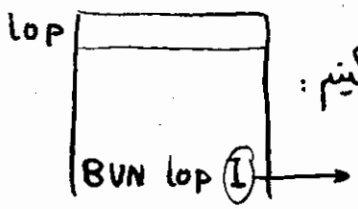
است.

سطرهای زبان اسمبلی حاوی ← دستورالعمل های ماشین
یا
شبه دستورها

کد	سمبل
V۸۰۰	CLA
⋮	⋮
V۰۰۱	HLT
⋮	⋮

به جای آدرس ها نیز از سمبل استفاده می کنیم : ADD 100 ⟶ ADD X

به جای مد (Mode) هم از سمبل استفاده می کنیم :



در این زبان یک سطری می تواند lab, (lable) داشته باشد یا نه (اختیاری).
(سمبل آدرس سطر)

ساختار یک سطر اسمبلی :

[lab,] (سمبل کاراکتری) (نوع عمل)
اختیاری

INP / Comment ← توضیح راجع به سطر که فقط برای برنامه نویس معتبر است.
CLA
ADD (آدرس سمبلیک)
Isz (b) (Lab) (b) I
خالی blank خالی

مثال

```
L1,  ADD X  I  /Comment
LY,  AND Y
     Isz CNT
X,   BUN L1
L۳,  CLA
     CMA  /Comment
```

ترجمهٔ مثال قبل برحسب Hex :

Hex	Hex
L1 = ۱۰۰	۹ ۱۰۳
L۲ = ۱۰۱	۰ ۱۰۵
	۶ ۱۰۴
X = ۱۰۳	۴ ۱۰۰
L۳ = ۱۰۴	۷ ۸۰۰
Y = ۱۰۵	۷ ۴۰۰

شبه‌دستورها

origin
ORG
END
HEX
DEC

```
ORG 100h
LDA X
ADD Y
STA Z
 ⋮
```

فرض کنیم برنامهٔ مقابل را داریم :

حال می‌خواهیم این برنامه ترجمه‌شده و از خط ۱۰۰ به بعد

قرار بگیرد. در این صورت از ORG 100h استفاده می‌کنیم.

ORG ، [lab,] نمی گیرد.

[lab,] ORG

ORG اپرند

آدرس سطر بعدی در حافظه اگر نیاز به حافظه داشته باشد.

شبه‌دستور END برای نشان دادن انتهای لیست برنامه به کار می‌رود.

```
ORG 100h
LDA X
ADD Y
STA Z
END
```

[lab,] نمی گیرد

END اپرند نمی گیرد

با HEX و DEC یک خانه رزرو شده و در آن ثابتی قرار می گیرد:

HEX ثابت هگزا [lab:]

DEC ثابت دسیمال [lab:]

در مثالی که ارائه و ترجمه شد، اگر قبل از آن ORG h۲۰۰ قرار دهیم ترجمه [lab:]ها عوض می شود و مثلاً ۲۰۰ = L۱, ۲۰۱=L۲ و ... می شوند.

Hex	Hex
L۱=۱۰۰	۹۱۰۳

ORG ۱۰۰H

L۱ , ADD ♿ X ♿ I /Comment

۱۶ بیت

O	R
G	♿
I	o
o	H
CR	LF
L	I
,	A
D	D
♿	X
♿	I
/	C

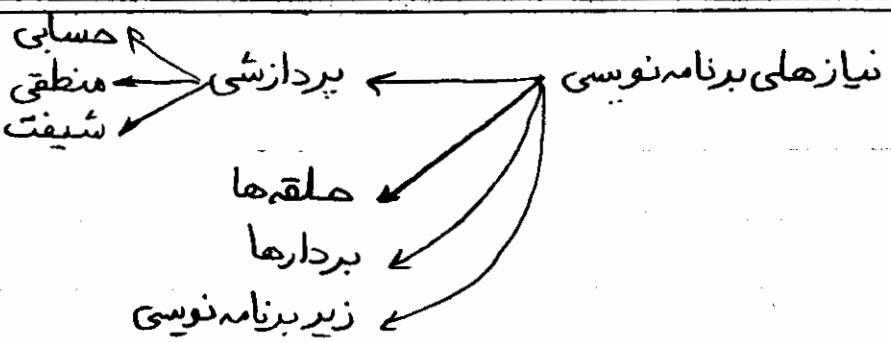
نحوه قرارگیری در حافظه

مربوط به عوض شدن سطر

X, Hex A۱۲۳

X, Hex -A۱۲۳

۱۰۱۰	۰۰۰۱	۰۰۱۰	۰۰۱۱
۰۱۰۱	۱۱۱۰	۱۱۰۱	۱۱۰۱



حسابی :

می‌خواهیم برنامه‌ای بنویسیم که حاصل X-Y را در خانه Z قرار دهد:

```
X, Hex -123A
Y, DEC -125
Z, Hex 0

ORG 100H
LDA Y / AC=Y
CMA
INC / AC=-Y
ADD X /AC=X-Y
STA Z
HLT
X, Hex -123A
Y, DEC -125
Z, Hex 0
END
```

آدرس Hex	حافظه HEX
۱۰۰	۲ ۱۰۷
۱۰۱	۷۴۰۰
۱۰۲	۷۰۸۰
۱۰۳	۱۱۰۶
۱۰۴	۳ ۱۰۸
۱۰۵	۷۰۰۱
۱۰۶	EDC۶
۱۰۷	FF۸۳
۱۰۸	۰۰۰۰

۱۲۳A ۰۰۰۱ ۰۰۱۰ ۰۰۱۱ ۱۰۱۰

-۱۲۳A ۱۱۱۰ ۱۱۰۱ ۱۱۰۰ ۰۱۱۰

۱۲۵

برای راحتی کار به جای مکمل ۲ها از مکمل ۱۶ها استفاده می‌کنیم. برای تبدیل عدد Hex به مکمل ۱۶، صفرهای سمت راست را حفظ کرده اولین رقم غیرصفر را از ۱۶ و بقیه را

از ۱۵ کم می‌کنیم. $-۱۲۳A \rightsquigarrow EDC۴$

	XH	XL
	۷۴۵	۴۳۲۱
دقت مضاعف	YH	YL

XL, Hex ۴۳۲۱
XH, Hex ۷۴۵

جمع با دقت مضاعف:

```
ORG  ۲۰۰H
LDA  XL
ADD  YL
STA  ZL
CLA
Cil
ADD  XH
ADD  YH
STA  ZH
HLT
```

```
LDA XH
ADC YH      → ADD with carry
STA ZH
```

منطقی: $X \vee Y = \overline{\overline{X} \wedge \overline{Y}}$

```
ORG  ۲۰۰H
LDA  X
CMA
STA  Z   /Z=X̄
LDA  Y
CMA
AND  Z   /AC=X̄∧Ȳ
CMA
STA  Z
HLT
X, Hex ...
Y, Hex ...
Z, Hex 0
END
```

شیفت منطقی

```
LDA X
CLE
cir یا cil
STA X
```

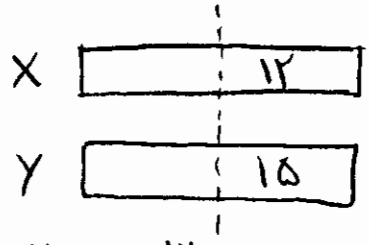
شیفت حلقوی

```
LDA X
cir یا cil
LDA X
cir یا cil
STA X
```

شیفت راست حسابی

```
LDA X
Cil
LDA X
Cir
STA X
```

ضرب : $P = X * Y$



چون ۱۴ bit * ۱۴ bit = ۳۲ bit لذا اعدادرا ۱۶ بیتی می‌گیریم.

X, Hex 12
Y, Dec 15

X می‌تواند مثبت یا منفی باشد ولی Y باید حتماً مثبت باشد. حال اگر Y منفی بود نخست آن را

```
lop, LDA  Y
     CIR
     STA  Y
     SIZE
     BUN  one
     BUN  zero
```

تست کنیم و در صورت منفی بودن علامت X و Y ، هردو را

```
one , LDA  P
      ADD  X
      STA  P
```
} جمع بادقت مضاعف

معکوس می‌کنیم و سپس ضرب می‌کنیم.

```
      CLE
ZER, LDA  X
      CIL
      STA  X
```
} شیفت بادقت مضاعف

```
      ISZ  CNT
      BUN  lop,
      HLT
X , Hex 12
Y , DEC 15
CNT, Dec -8        → -14
P , Hex 0
      END
```

برای ضرب دو عدد ۱۶ بیتی دهم باید

تغییرات مقابل را انجام دهیم:

شیفت بادقت مضاعف

```
CLE
LDA  XL
CIL
STA  XL
LDA  XH
CIL
STA  XH
```

باید تست کنیم که

اگر X_L منفی بود X_H را با FFFF پر کنیم.

تکالیف فصل ۶ : ۶ ، ۸ ، ۱۱ ، ۱۲ ، ۱۹ ، ۲۲ ، ۲۸ + مسئله کنترل عامل وقفه .

نیاز به بردارها :

```
      ORG  100 H
A,    Hex 1      A[0]
      Hex 2      A[1]
      ⋮          ⋮
      Hex 9      A[8]
      Hex A      A[9]
```

بردار A را به شکل مقابل تعریف می‌کنیم که آدرس اولین عنصر ما است که با سمبل A نشان داده می‌شود. بردار مجموعهٔ عناصر متوالی در حافظه است ، که فقط عنصر اول اسم دارد. در مورد بردار دو نکته مطرح می‌شود: یکی آدرس شروع بردار دیگری طول بردار.

```
AA ,  Hex  100      آدرس بردار A
LA ,  Dec  10       طول بردار A
```

می‌خواهیم حلقه‌ای تشکیل دهیم که در هر تکرار یک عنصر از بردار را استفاده کنیم :

به این منظور از خانه‌ای از حافظه به نام pt (Pointer) استفاده می‌کنیم . از خود AA استفاده نمی‌کنیم چون نباید تغییر کند .

```
CNt,  Hex .
pt ,  Hex .
      LDA  AA
      STA  pt
      LDA  LA
      CMA
      INC
      STA  CNt
lop,  ADD  pt I
      Isz  pt / INC pt   بجای ⟷ { STA SUM
                                 LDA pt
                                 INC
                                 STA pt
                                 LDA SUM
      Isz  CNt
      BUN  lop
      STA  SUM
      HLT
```

حال می‌خواهیم بدون استفاده ازمد غیرمستقیم [1100]

```
lop,  ADD  A
      Isz  lop
      Isz  cnt
      BUN  lop
```

این کار را انجام دهیم:

H 001

A, H9x 1 [illegible]

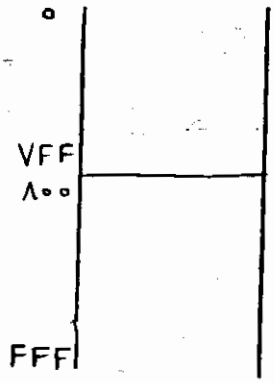
استفاده ازمدغیرمستقیم بهتر است. چون درمد مستقیم محتوی دستور درطول اجرا

عوض می‌شود واین نامطلوب است. چون دنبال کردن یک برنامه برای یافتن خطای

آن بسیار مشکل خواهدبود. لذا درسیستم های محافظتی در پروسسور تعبیه

می‌شود تا هنگام اجرای دیتا یا برداشت دستورالعمل نامطلوب سیگنال پیام

تولیدشود.



زیربرنامه‌نویسی ← لینک آدرسها ← call, RET

لینک پارامترها- مبادلهٔ اطلاعات ← متغیرسراسری، رجیسترها، آدرس بلوک پارامترها

آدرس بلوک پارامترها ← رجیستر، ازطریق BSA

مبادلهٔ اطلاعات :

```
      ORG  50H
       ⋮
55    BSA  CM2     ← 100
       ⋮

      ORG  100h
CM2,  Hex 0        ← محل ضبط آدرس برگشت که ۵۶ است.
       ⋮
      BUN  CM2  I
```

نحوهٔ رفت و برگشت :

متغیر سراسری : global variable

```
 ORG   50H
{LDA   A
 STA   X
55 BSA   CM2
{LDA   CX
 STA   CA
 BSA   CM2
  ⋮
A, Hex 0
CA,
 ⋮
CB,

  ORG  100H
CM1, Hex 0
{LDA  X
 CMA
 INC
 STA  CX
 BUN  CM2  I
```

محل مبادلهٔ اطلاعات یک متغیر سراسری

باید باشد که بتوان از هر زیر برنامه به آن

دسترسی داشت به این ترتیب که از متغیر محلی

مقدار را به متغیر سراسری انتقال می دهیم و به

زیر برنامه مراجعه کرده و توسط زیر برنامه متغیر

سراسری تغییر کرده و سپس جواب را از این متغیر

به متغیر محلی انتقال می دهیم.

استفاده از چنین متغیر سراسری مطلوب نمی باشد چون

دنبال کردن برنامه باز هم مشکل خواهد بود.

این روش در پاسکال به شکل مقابل است:

```
X := A ; CMY ;  CA := CX ;
X := B ;  CMY ;  CB := CX ;
```

(CMY; procedure)

روش رجیستری:

حال می‌خواهیم از روش رجیستری استفاده کنیم و مبادلهٔ اطلاعات داشته باشیم

در این حالت فرم پاسکال آن به شکل زیر است:

```
CA := CMY(A) ;
CB := CMY(B) ;
```

```
LDA   A
BSA   CMY
STA   CA
  ⋮
LDA   B
BSA   CMY
STA   CB
  ⋮
ORG   100
CMY, Hex 0
     CMA
     INC
     BUN CMY I
```

این روش، روش خوبی است. فقط مشکلی که وجود دارد این است که تعداد رجیسترها محدود است.

اگر پارامترها بیش از تعداد رجیسترها باشد باید از روش آدرس بلوک پارامترها استفاده کنیم.

آدرس بلوک پارامترها:

می‌خواهیم برنامه‌ای بنویسیم که OR را انجام دهد. پارامترها (بلوک)

```
10   BSA OR
11  [A, Hex 1234
12   B, Hex F5AB
13   DAB, Hex 0]
14   دستور
  ⋮
20   BSA OR
21  [C, Hex ...
22   D, Hex ...
23   DCD, Hex 0]
24   دستور
```

(بلوک پارامترها)

روش مقابل آدرس بلوک پارامترها از طریق BSA است.

```
   ORG  100 h
OR, Hex 0
   LDA  OR I / CیاA
   CMA
   STA  T
   ISZ  OR
   LDA  OR I /DیاB
   CMA
   AND  T
   ISZ  OR
   STA  OR I
   ISZ  OR
   BUN  OR I-1
T, Hex 0
```

اگر ISz OR در آخر برنامه قرار ندهیم

با BUN به خط ۱۳ یا ۲۳ می رود و ما می خواهیم

به ۱۴ یا ۲۴ برویم تا بتوانیم دستور بعدی را

Fetch و اجرا کنیم .

مشکل این روش این است که اگر بلوک پارامترها

طولانی یا بردار باشد باید مرتب از ISz OR استفاده کنیم

تا به آخر برسیم و برنامه طولانی می شود، لذا از مناسب ترین روش که بلوک پارامترها

با رجیستر است استفاده می کنیم :

```
برنامه:
   LDA   AB1 / AC=100
   BSA   OR
   دستور
   :
   LDA   AB2 / AC=103
   BSA   OR
   دستور
   :
   BUN   OR I
pt, Hex 0   / 100 یا 103
```

```
زیربرنامه:
   ORG  200 h
OR, Hex 0
   STA  pt
   LDA  PT I
   CMA
   STA  T
   ISZ  pt
   LDA  PT I
   CMA
   AND  T
   CMA
   ISZ  PT
   STA  PT I
```

```
   ORG  100
A, Hex ---
B, Hex ---
OAB,
C,
D,
OCD,
AB1, Hex 100
AB2, Hex 103
```

بلوک پارامترها

در مورد کار با زیربرنامه‌ها، خود برنامه را در بلوک پارامتر قرار نمی‌دهیم بلکه آدرس شروع آن را فقط در بلوک پارامترها قرار می‌دهیم.

مثال: جمع عناصر یک بردار:

```
        ORG 200
SUM,    Hex 0      /AC=40 یا 42
        STA pt
        LDA pt I   /AC=50 یا 90
        STA AR
        ISZ pt
        LDA pt I   /AC=10 یا 20
        CMA
        INC        /AC=-10 یا -20
        STA CNT
        CLA
Lop,    ADD AR I
        ISZ AR
        ISZ CNT
        BUN Lop
        BUN SUM I
```

```
B1,  Hex 40
B2,  Hex 42

LDA  B1
BSA  SUM
STA  SA
 ⋮
LDA  B2
BSA  SUM
STA  SB
 ⋮
```

(نحوه مراجعه)

```
        ORG 40
AA,  Hex 50      (بلوک ۱)
LA,  Dec 10
AB,  Hex 90      (بلوک ۲)
LB,  Dec 20
        ORG 50
A,   Hex 1
 ⋮
        ORG 90
B,   Hex 2
 ⋮
```

```
pt,  Hex 0   / 40 یا 42
AR,  Hex 0   / 50 یا 90
CNT, Hex 0   / -10 یا -20
```

تمرین: اگر نحوهٔ مراجعه به شکل زیر باشد مسئله را حل کنید:

(نحوه مراجعه)

```
BSA  SUM
Hex  40
STA  SA
 ⋮
BSA  SUM
Hex  42
STA  SB
 ⋮
```

ورودی/خروجی:

(ورودی)

```
LI,  SKI
     BUN LI
*    INP
     OUT
```

اشکالی که در * وجود دارد این است که ممکن است خروجی آماده نباشد و باید چک کنیم که خروجی آماده است یا نه.

(خروجی)

```
LO,  SKO
     BUN LO
     OUT
```

```
ORG 0
ISR, Hex 0
     STA TA
     Cir
     STA TE
     SKI
     BUN OT
     INP
     STA IB
     SKO
     BUN RT
OT , LDA OB
     Out
RT , LDA TE
     Cil
     LDA TA
     ION
     BUN ISR I
TA , Hex 0
TE , Hex 0
IB , Hex 0
OB , Hex 0
```

در روتین وقفهٔ مقابل مواردی که دور آنها خط کشیده شده زائد هستند و می‌توانند حذف شوند.

TE, TA باید محلی و IB, OB باید سراسری باشند.

اشکالی که این روتین وقفه دارد این است که با فرودی و خروجی یکی خانه است و در نتیجه اطلاعات را بر روی هم می‌گذارد و لذا اطلاعات قبلی از بین می‌رود و فقط اطلاعات آخر باقی می‌ماند.

در صورتیکه باخرها باید دارای چند خانه باشند (بردار باشند).

اصلاح:

```
IB, Hex 0   ~~~>   PIB, Hex 100
OB, Hex 0          POB, Hex 200
```

(در خط ۹): `STA IB` ~~~> `STA PIB I` ، `ISZ PIB`

(در خط ۱۲): `LDA OB` ~~~> `LDA POB I`

و در خط ۱۳ بعد از Out باید ISZ POB قرار بگیرد.

مثال: زیر برنامه‌ای بنویسید که محتوی AC را ۴ تا شیفت منطقی چپ داده و در جای خود قرار دهد.

راه اول:

```
ORG 100
SHF, Hex 0
     Cil
     Cil
     Cil
     Cil
     AND MSK
     BUN SHF I
MSK, Hex FFF0
```

راه دوم:

```
SHF, Hex 0
lop, CLE
     Cil
     ISZ CNT
     BUN lop
     BUN SHF I
CNT, Dec -4
```

راه دوم غلط است. در اولین بار مراجعه درست انجام می دهد اما در مراجعه های بعدی مقدار CNT دیگر ۴- نیست بلکه صفر است.

```
SHF, Hex 0
     STA T
     LDA CST
     STA CNT
     LDA T
lop, CLE
     CIL
     ISZ CNT
     BUN lop
     BUN SHF I

CNT, Dec 0
CST, Dec -4
T, Hex 0
```

اصلاح راه دوم :

مثال :

```
     ORG 200
INY, Hex 0
LI,  SKI
     BUN LI
     INP
     BSA SHF
     BSA SHA
LY,  SKI
     BUN LY
     INP
     BUN INY I

     ORG 300
LOP, BSA INY
     STA PIB I
     ISZ PIB
     ISZ CNT
     BUN lop

PIB, Hex F00
CNT, Dec -100
```

زیر برنامه بنویسید که دو کاراکتر را گرفته و آنها را کنار یکدیگر قرار دهد.

Assembler (مترجم) :

مترجم از یکی سری ضرایب ثابت استفاده می کند که جداول کدها هستند.

جدول NMR None Mem. Reference

سمبل	کد
CLA	V800
CMA	V400
⋮	⋮
HLT	V001
INP	F800
⋮	⋮
IOF	F040

MR

سمبل	کد
AND	0000
ADD	1000
⋮	⋮
ISZ	6000

شبه دستورها

ORG	آدرس
END	
Hex	روتین
Dec	

جدول دیگری به نام جدول آدرس های سمبلیک وجود دارد.

		آدرس Hex	محتوا Hex	
	ORG ۲۰۰h			
INP,	Hex ۰	۲۰۰	۰۰۰۰	
LI,	SKI	۲۰۱	F۲۰۰	
	BUN LI	۲۰۲	۴۲۰۱	۴۰۰۰ + ۰۲۰۱
	INP	۲۰۳	F۸۰۰	
	BSA SHF	۲۰۴	۵۱۰۰	
	BSA SHAF	۲۰۵	۵۱۰۰	مرور دوم
LP,	SKI	۲۰۶	F۲۰۰	
	BUN LP	۲۰۷	۴۲۰۶	
	INP	۲۰۸	F۸۰۰	
	BUN INP I	۲۰۹	C۲۰۰	۴۰۰۰ + ۰۲۰۰ + ۸۰۰۰ ← مربوط I

در مرور اول جدول آدرس های سمبلیک بدست می آید.

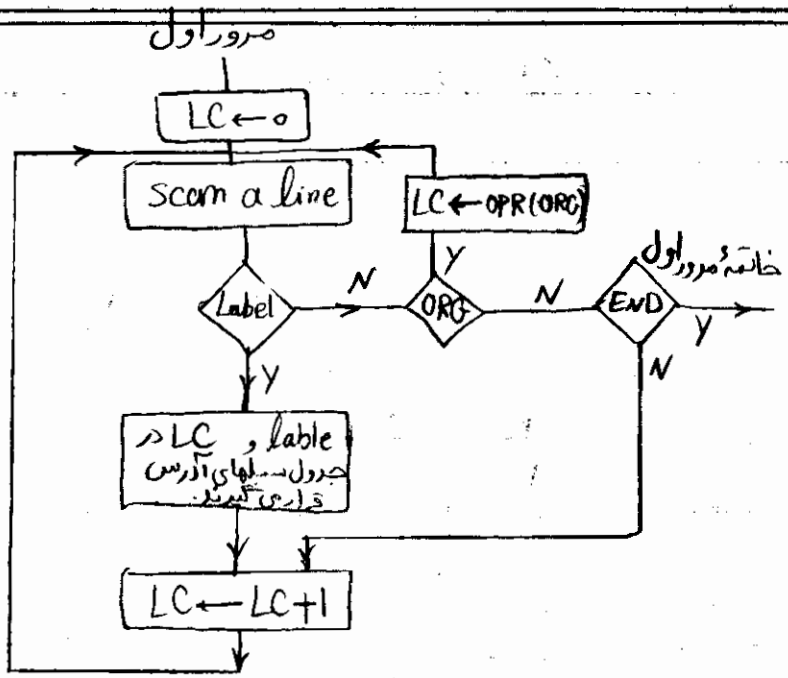
سمبل	مقدار
INP	۲۰۰
LI	۲۰۱
LP	۲۰۶

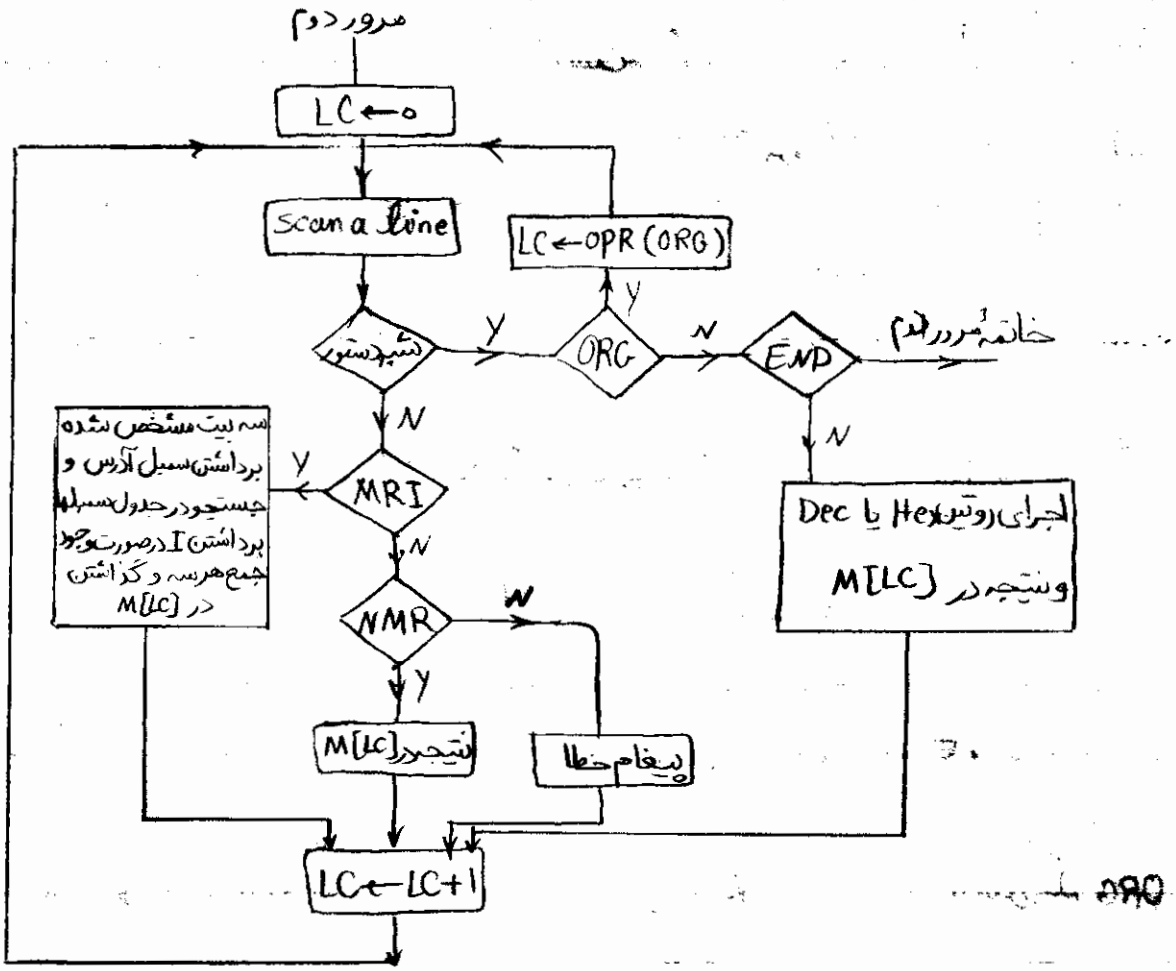
مرور اول

در مرور دوم هر سطر ترجمه می شود.

حال دو مرور را به شکل بلوک دیاگرام نشان می دهیم:

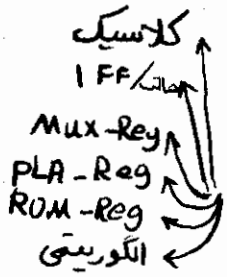
LC متغیری است که در آن شمارهٔ کلمه ای که در سطر باید ترجمه شود در آن قرار می گیرد. اگر ORG نداده باشیم مقدار اولیهٔ LC صفر خواهد بود. در scan a line یک سطر را بررسی دارد. بعد چک می کند که آیا lable دارد یا نه. به ازای ORG و END، LC اضافه نمی شود. در مرور دوم بعد از ترجمه اطلاعات از خانه صفر حافظه قرار می گیرند (در صورتیکه ORG نداشته باشیم)


مرور اول
LC ← 0
Scan a line
LC ← OPR(ORG)
Label
N
Y
ORG
N
Y
END
N
Y
خاتمه مرور اول
LC و lable در جدول سمبلهای آدرس قرار می گیرند
LC ← LC+1


مرور دوم
LC ← 0
Scan a line
LC ← OPR (ORG)
شبه دستور
Y
ORG
N
Y
END
N
خاتمه مرور دوم
N
MRI
Y
سه بیت مشخص شده برداشتن سمبل آدرس و جستجو در جدول سمبلها برداشتن I در صورت وجود جمع هر سه و گذاشتن در M[LC]
N
NMR
N
Y
اجرای روتین Hex یا Dec ونتیجه در M[LC]
نتیجه در M[LC]
پیغام خطا
LC ← LC+1

فصل ۷:

میکروپروگرامینگ

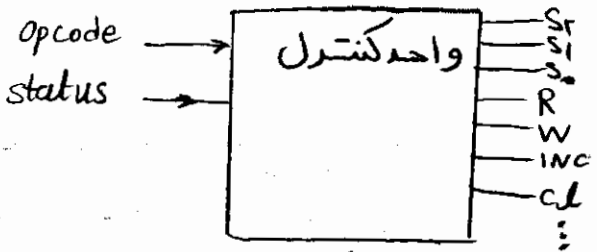


کنترل ← سخت‌افزاری

کنترل ← میکروپروگرامینگ

در روش‌های سخت افزاری با تغییر دستورها سخت افزار تغییری کند درحالیکه درمیکروپروگرامینگ

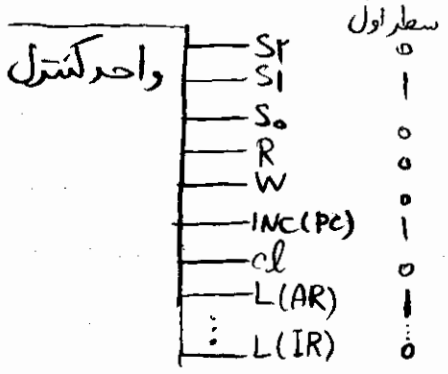
روش نرم افزاری است و تا آخرین لحظه با تغییر دستورها واحد کنترل تغییر نمی کند.



فرض کنیم حدود ۴۰ خروجی داریم. تعداد ترکیبهای مختلف ۰ و ۱ها برای ۴۰ خروجی

برای این کامپیوتر مطمئناً بسیار کمتر از 2^{40} خواهد بود (تعداد این حالتها برابر تعداد

سطرهای RTL است) (حدود ۵۰ تا) به عنوان مثال: ← یک کلمهٔ کنترل



سطر اول $R'T_0$: $AR \leftarrow PC$, $PC \leftarrow PC+1$

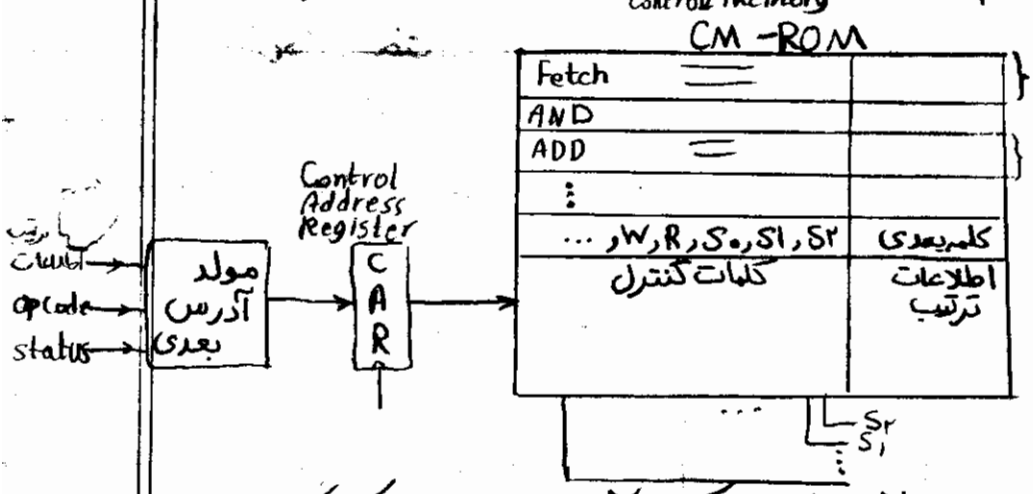
$R'T_1$: $IR \leftarrow M$

هر سطر RTL اگر مشروط نباشد یک کلمهٔ کنترلی را تولید می‌کند. مشروط‌ها دوکلمه می‌توانند

داشته باشند: if X then μ_1, μ_2 else μ_3, μ_4

در میان سطح‌های مشروطی هم تعدادی از آنها در یک حالت کاری انجام نمی دهند. از طرفی حالت‌های تکراری داریم. لذا در نهایت تعداد حالتها از ۵۰ هم کمتر خواهد شد.

حال یک حافظه تعریف می‌کنیم به شکل زیر:



در قسمت اطلاعات ترتیب، ترتیب آمدن کلمات کنترل آمده است. به این شکل که در قسمت اطلاعات ترتیب هر خانه از حافظه محل کلمهٔ بعدی مشخص شده است.

به مجموعهٔ اطلاعات ترتیب و کلمات کنترل ریز دستور μ-instruction می گوییم.

به مجموعهٔ میکرو روتین ها، میکرو پروگرامینگ می گوییم.

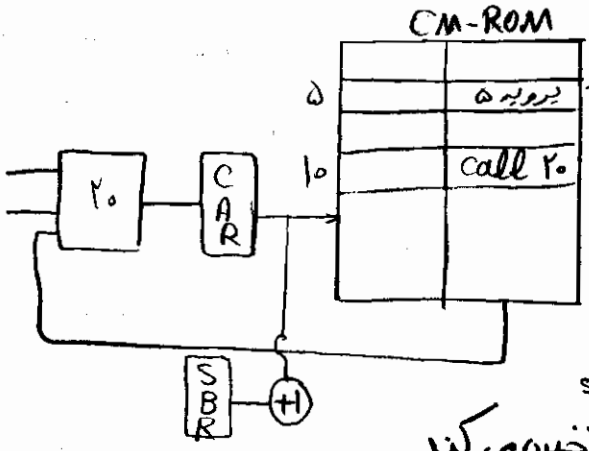
در این روش تعداد بیتهای هر کلمه در یک خانه زیاد است ولی تعداد کلمه ها کم است. لذا آدرس ها هم کوتاه خواهند بود و تلفات کمتر می‌شود.

با توجه به فصل ۵، ۲۸ تا میکرو روتین خواهیم داشت.

وقتی کامپیوتر را روشن می‌کنیم اولین کلمهٔ Fetch باید در خروجی قرار گیرد. یعنی مقدار اولیه CAR صفر است. با آمدن کلاک، اولین کلمهٔ Fetch اجرا شده و با توجه اطلاعات ترتیب مقدار بعدی CAR می‌شود. لذا CAR ترتیب اجرای کلمات را تعیین می‌کند.

ترتیب دستورات	ترتیب ریز دستورات
نقطه شروع	نقطه شروع
توالی	توالی
انشعاب مشروط	انشعاب غیر مشروط
انشعاب غیر مشروط	انشعاب مشروط
RET, Call	RET, Call
	Map

در قسمت مولد آدرس بعدی، همواره باید یک آدرس (آدرس بعدی) آماده باشد. ورودی‌های آن بیت‌های اطلاعات ترتیب و status ها و opcode است.



در مورد دستور Call به عنوان مثال:

10 Call ۲۰

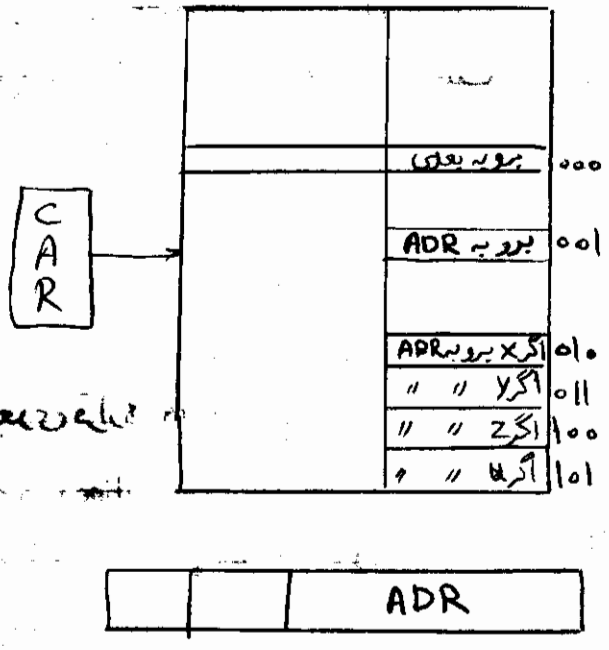
آدرس ۲۰ را در مولد آدرس بعدی قرار داده و آدرس ۱۱ را در یک رجیستر به نام SBR ذخیره می‌کند.

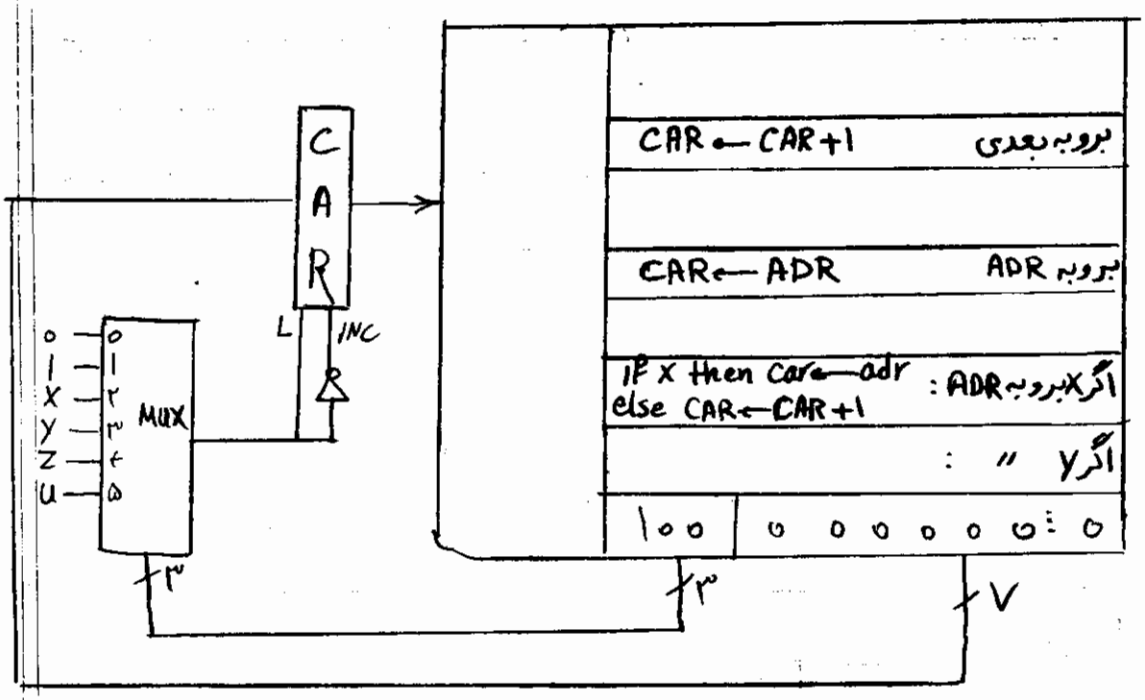
حال سوال زیر مطرح می‌شود:

درمورد دستور Fetch در سومین μ-ins آن چه چیزی باید قرار بگیرد. در این حالت ۲۵ انشعاب داریم که باید یکی انتخاب شود. در نتیجه مفهومی به نام Map مطرح می‌شود. Map یعنی تولید آدرس ریز دستور از میان چند انشعاب.

تفاوت CM-ROM با روش ROM-Reg این است که در ROM تعداد کلمات بسیار زیاد است و همچنین قرار گرفتن اطلاعات در آن بطور پراکنده است و نظمی ندارد و در نتیجه نمی توان جای کلمات را عوض کرد. درحالیکه در CM-ROM می توانیم بسیکل یک دستور را به راحتی تغییر دهیم.

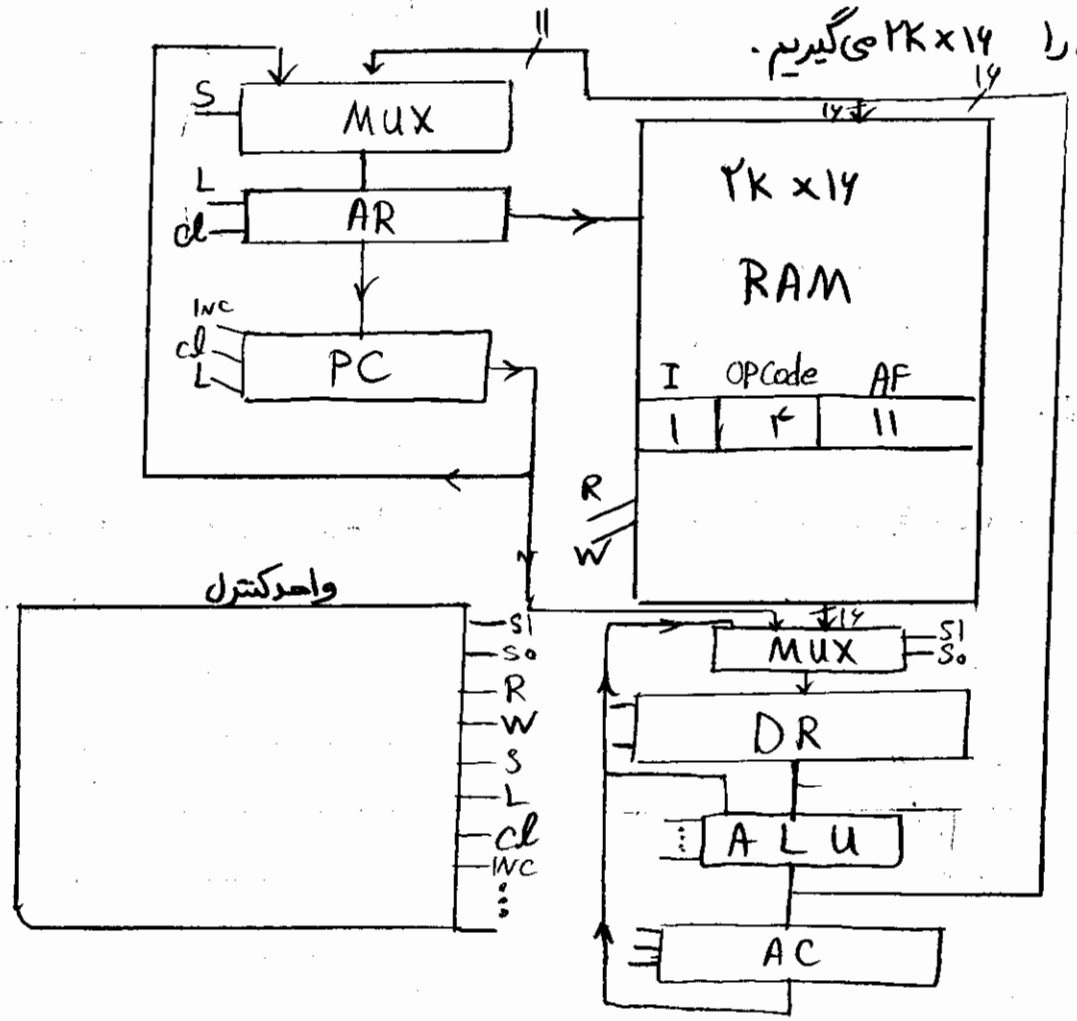
مثال:





طراحی کامپیوتر فصل ۵:

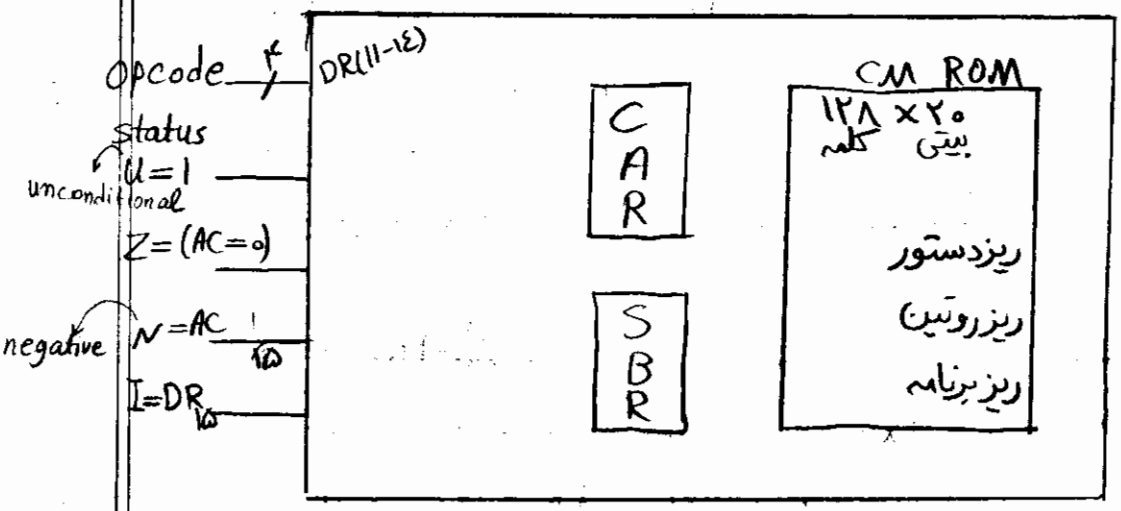
حافظه را ۲K x ۱۶ می‌گیریم.



هرزمانکه خواهیم بخوانیم یا بنویسیم از طریق DR فقط امکان پذیر است.

تعداد MUXهای استفاده شده ۲۷ = ۱۱ + ۱۶ است.

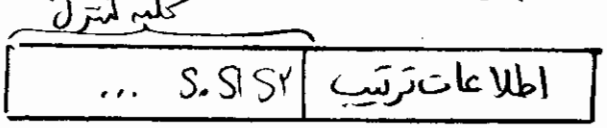
ADD	0000	EAC ← AC + M[EA]
STA	0001	M[EA] ← AC
BPA	0010	if AC>0 then PC ← EA
⋮	⋮	⋮

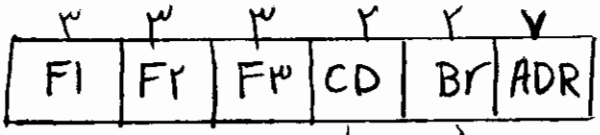


دراین روش تعداد call های تودرتو در برنامه (در RAM) با یک SBR می‌تواند هر چندتا باشد و هیچ ربطی به تعداد SBR ندارد. (چون آدرس های برگشت در حافظه RAM ذخیره می‌شود و ربطی به SBR ندارد)

هروقت U را به عنوان شرط انتخاب کنیم چون همواره ۱ است شرط انجام می‌شود.

فرمت افقی μ-ins :





D

درمورد بیت های Br :

JMP	۰ ۰
CALL	۰ ۱
RET	۱ ۰
Map	۱ ۱

و درمورد CD :

U	۰ ۰
Z	۰ ۱
N	۱ ۰
I	۱ ۱

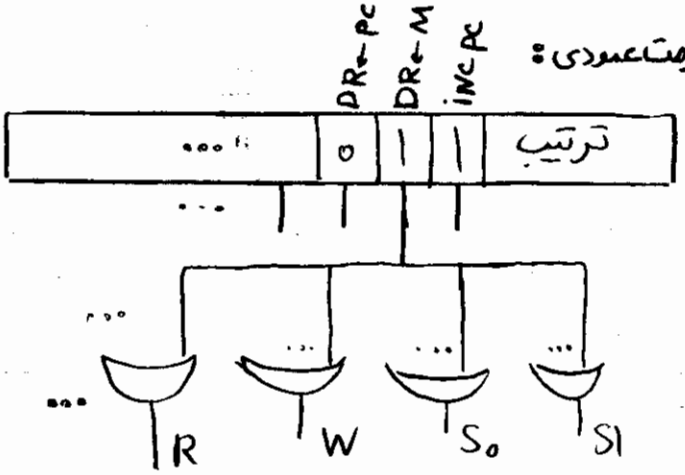
توالی را یک حالت خاص از انشعاب غیر مشروط می گیریم.

در فرمت افقی طول کلمه بلند، اکثر بیت ها صفر ولی امکان عملیات موازی فراهم است.

لذا تلفات حافظه زیاد است. برای جلوگیری از تلفات روشهای مختلفی وجود دارد :

یک راه این است که به جای

فرمت عمودی :

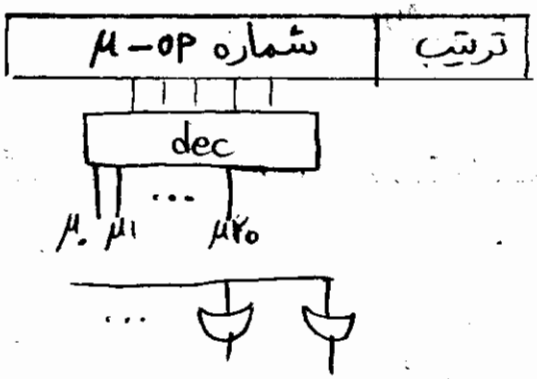


هر μ-op یک بیت در نظر بگیریم :

فرمت افقی
- طول زیاد
- استفاده غیر موثر از حافظه
- عملیات موازی
- dec ندارد.

فرمت عمودی تر

راه دیگر :



باقیماندهٔ مباحث اطلاعات ترتیب:

Br		
۰ ۰	JMP	if (CD) then (CAR ← Adr) else (CAR ← CAR + 1)
۰ ۱	call	if (CD) then (SBR ← CAR+1, CAR ← Adr) else (CAR ← CAR+1)
۱ ۰	RET	CAR ← SBR
۱۱	Map	CAR ← f(op-Code)

Mapping در انتهای سیکل fetch باید اجرا شود.

fetch	U JMP ۱	AR ← PC , PC ← PC+1
	U JMP ۲	DR ← M
	Map	CAR ← f(op-Code)

پیاده سازی Map

حافظه کنترل

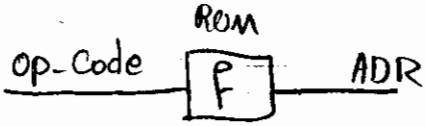
۰	fetch
۵	ADD
۱۰	ISZ
۲۰	AND

۱۹ آدرس

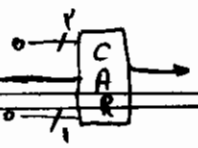
Map

آدرس CM	Op
۰ ۰ ۰ ۰ ۱ ۰ ۱	۰ ۰ ۰ ۰
۰ ۰ ۰ ۱ ۰ ۱ ۰	۰ ۰ ۰ ۱
۰ ۰ ۱ ۰ ۱ ۰ ۰	۰ ۰ ۱ ۰
۰ ۰ ۱ ۱ ۱ ۰ ۰	۰ ۰ ۱ ۱

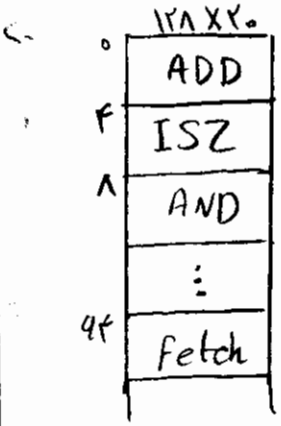
یک روش برای Mapping داشتن یک مدار ترکیبی مانند ROM است که جدول بالا را تولید کند.



روش دیگر این است که تابع f را به صورت زیر تعریف کنیم:



۴ یعنی دو صفر در سمت راست و یک صفر در سمت چپ op code قرار بگیرد. در این صورت نیازی به مدار ترکیبی نداریم. 11 Map CAR ← 0 (op-Code) 0 0

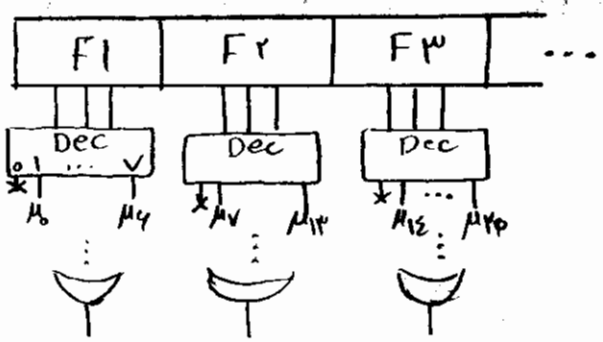


بدین قراردادی نحوهٔ قرارگیری سیکل ها در حافظهٔ کنترل به صورت مقابل است:

سیکل fetch در اینجا در خانهٔ ۶۴ قرار دارد.

در این روش Mapping، برای هر دستور ۴ خانه وجود دارد و ۶۴ خانهٔ خالی در انتهای حافظهٔ کنترل قرار دارد. اگر دستوری از ۴ خانه بیشتر نیاز داشت در انتهای حافظهٔ کنترل قرار می گیرد و با U JMP می توان به آن رجوع کرد.

ادامهٔ بحث کلمهٔ کنترل:



اگر همهٔ سیکل ها را بنویسیم و μ-op ها را جدا کنیم نهایتاً ۲۱ μop خواهیم داشت که آنها را شماره گذاری می کنیم:

μ_0 $DR \leftarrow M$

μ_1 $AR \leftarrow DR$

$\vdots$

μ_{20} $PC \leftarrow PC+1$

در فرمت عمودی تر از این روش استفاده شده است. مشکلات این فرمت این است که سطوح تأخیر بیشتر شده و عملیات موازی نداریم. یعنی همزمان دو μ-op نمی توانند فعال شوند، و برای انجام همزمان دو μ-op باید دو کلاک استفاده کنیم. برای اینکه بتوانیم عملیات موازی داشته باشیم ۲۱ μ-op را به ۳ گروه ۷تایی تقسیم می کنیم و ۳ گروه را دکود می کنیم در این صورت کلمهٔ کنترل 3×3 خواهد شد. در روش قبل کلمهٔ کنترل ۵ بیتی بود که پس از دکود شدن ۲۱ μ-op را به ما می داد. ۹ بیت کلمهٔ کنترل حد وسط است و امکان عملیات موازی را برای ما فراهم می کند.

سوال: اگر ۲۴ μ-op داشتیم نمی توانستیم از فرمت ۹ بیتی استفاده کنیم. چرا؟

چون در حالت ۲۱ μ-op اگر می خواستیم هیچ μ-op (در هر گروه) اجرا نشود کد ۰ را انتخاب می کردیم که از ترمینال صفر dec هم استفاده نکرده بودیم. حال اگر ۲۴ μ-op را به ۳ گروه ۸تایی تقسیم کرده و دکود کنیم در هر حال یکی از خروجی های dec مقدار یکعادی حالتی وجود نخواهد داشت که هیچ μ-op اجرا نشود.

نکته: micro-op های هرگروه باید به نحوی انتخاب شوند که در هیچ دستوری همزمان اجرا نشده باشند (یعنی دو micro-op از یک گروه انتخاب نشوند)

جدول F1

کد	سمبل	عمل
000	Nop	——
001	cl AC	AC ← 0
010	read	DR ← M[AR]
011	write	M[AR] ← DR

جدول F2

کد	سمبل	عمل
000	Nop	——
001	MADD	AC ← AC + DR
010	MXOR	AC ← AC ⊕ DR
⋮		

لذا با سمبل های بالا یک زبان سمبلیک خواهیم داشت:

زبان سمبلیک:

```
        ORG  64
[lab:]  F1 , F2 , F3  U JMP  Lab   } آدرس
                      Z call Lab   }
                      N
                      I
                      U RET  ——    بدون آدرس
                        MAP ——     فقط با U
```

(F1, F2, F3: سمبل؛ برای سطرهای متوالی ← Next)

سطرهای متوالی چون با U JMP انجام می‌شود لذا هر سطر باید Lab داشته باشیم

برای اینکه تعداد Lab ها زیاد نشود در سطرهای متوالی از کلمهٔ Next در قسمت آدرس استفاده می‌کنیم.

ترتیب نوشتن F3, F2, F1 در زبان سمبلیک اهمیتی ندارد. فقط باید دقت کنیم که از هر یک گروه فقط یک micro-op انتخاب شود.

نوشتن میکروپروگرام:

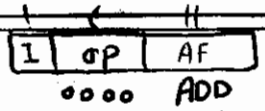
```
        ORG 4F
Fetch : PCTAR, INC PC   U JMP Next    AR←PC, PC←PC+1
        read            U JMP Next    DR←M
        DRTAR           I JMP IND     AR←DR
        ORG 20
IND :   read            U JMP Next    DR←M
```

مشکل دارد

اجرای سیکل IND باعث می‌شود که کلمهٔ دیگری از حافظه خوانده شده و در DR قرار بگیرد که این کلمه محتوی آدرس اپرند است امّا این کار باعث می‌شود که op-code ای که در سیکل Fetch خوانده شده و در DR بود از بین برود. (از مشکلات نداشتن IR). لذا نخست Map را انجام می‌دهیم و سپس IWD را.

```
        ORG 4F
Fetch : PCTAR, INC PC   U JMP Next
        read
        DRTAR           U Map

        ORG 0
ADD :   Nop             I JMP IND
LI  :   read            U JMP Next    DR←M
        MADD            U JMP Fetch   DR←DR+AC
        ORG 47
IWD :   read            U JMP Next
        DRTAR           U JMP Next    LI
        read            U JMP Next
        MADD            U JMP Fetch
```



0000 ADD
0001 STA
0010 BPA

STA ADR [I] M[EA] ← AC

BPA adr [I] if AC>0 then pc ← EA

```
        ORG 4
STA :   NOP    I  CALL IND
        ACTDR  U  JMP  Next
        Write  U  JMP  Fetch
```

```
        ORG 8
BPA :   NOP    I  CALL IND
        Nop    Z  JMP  Fetch
        Nop    N  JMP  Fetch
        ARTPC  U  JMP  Fetch
```

مشاهده می‌کنیم که برای BPA نمی‌توان تعداد کلاک‌ها را معین کرد.

درحالت	مستقیم	غیرمستقیم
=	۵ = ۰	۷
	۶ < ۰	۸
	۷ > ۰	۹

درمورد BPA می‌توان خط اول را که I را تست می‌کند عقب‌تر انداخت چون DR به هم نمی‌خورد.

```
        ORG 8
BPA :   NOP    Z  JMP  Fetch
        Nop    N  JMP  Fetch
        Nop    I  CALL IND
        ARTPC  U  JMP  Fetch
```

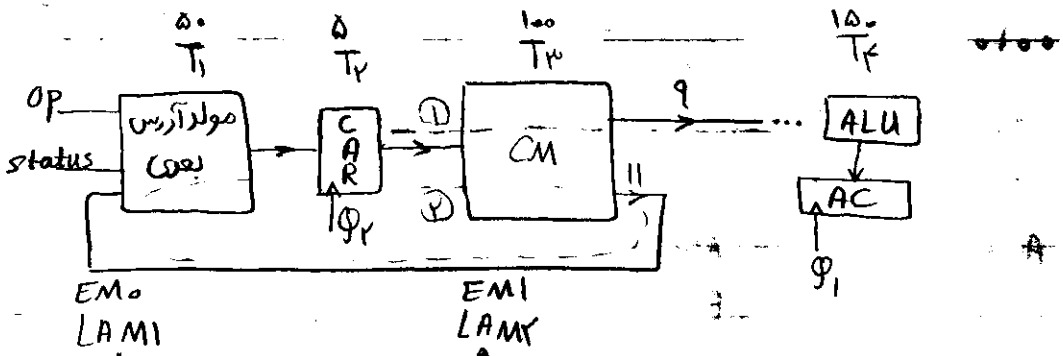
درنتیجه:

در این حالت:

مستقیم	غیرمستقیم
۴ = ۰	۴
۵ < ۰	۵
۷ > ۰	۹

با مثالهای زده شده می‌توان انعطاف و سادگی میکروپروگرامینگ را دید. امّا اشکالی که نسبت به حالت سخت‌افزاری دارد این است که تعداد کلاکها بیشتر شده و همچنین چون

از ROM استفاده کرده‌ایم نسبت به گیت‌ها دارای تأخیر بیشتری خواهد بود.



تأخیرات مربوط به مسیر ۱ : $T_2 + T_3 + T_4$ ↑ ۲۵۵ ↑ LAM۱

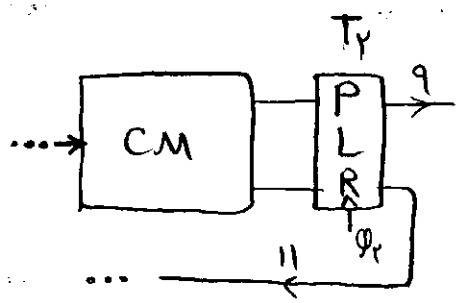
$f_{max} = \frac{1}{۲۵۵}$

تأخیرات مربوط به مسیر ۲ : $T_2 + T_3 + T_1$ ↑ ۱۵۵ ↑ LAM۱

LAM۱ ≡ load address Microinstruction 1

EM۰ ≡ Execute " "

برای جلوگیری از این تأخیرات که به صورت سری انجام می‌شوند از یک رجیستر به نام PLR (Pipe line Reg.) استفاده می‌کنیم. کار این رجیستر این



است که Microinstruc. را در خود نگه می‌دارد و باعث می‌شود که تأخیرها موازی با هم طی شوند. یعنی در طول زمانیکه تأخیر T_4 طی می‌شود تأخیر T_3 هم انجام می‌شود.

کلاک‌های φ_1 و φ_2 هم فرکانس هستند ولی با هم اختلاف فاز دارند.

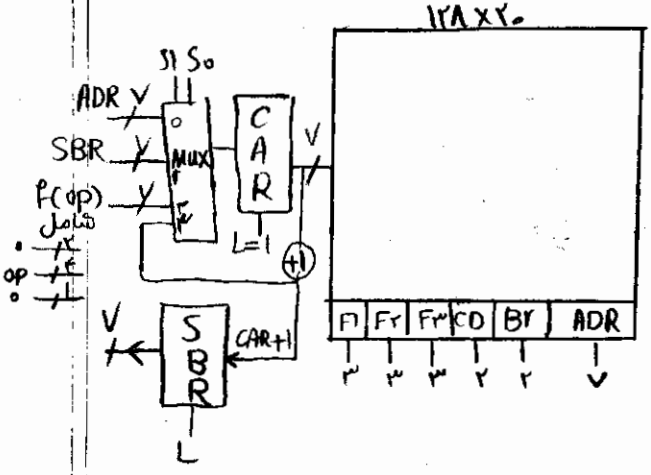
ORG ۶

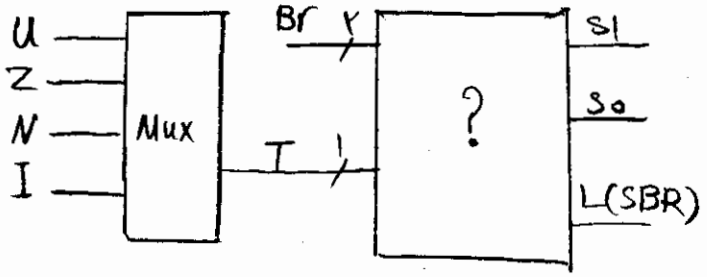
```
AND : Nop      I   JMP  INDY
LY  : read     U   JMP  Next
      MAND     U   JMP  Fetch
      ORG 69
INDY: read     U   JMP  Next
      DRTAR    U   JMP  LY
```

مشاهده کنیم که با این روش باید برای هر دستور یک سیکل IND مجزا داشته باشیم این مشکل

با وجود CALL , RET حل می‌شود:

```
      ORG 0
ADD : Nop   I  CALL IND
      read  U  JMP  Next
      MADD  U  JMP  Fetch
      ORG 97
IND : read  U  JMP  Next
      DRTAR    RET
```





	Br	T	SI	S_0	L(SBR)
JMP	0 0	0	1	1	0
	0 0	1	0	0	0
call	0 1	0	1	1	0
	0 1	1	0	0	1
RET	1 0	α	0	1	0
Map	1 1	α	1	0	0

مدار مجهول (؟) را

با داشتن جدول مقابل

به راحتی پیاده سازی

می کنیم.

سوال: برای اجرای دستور ADD چند کلاک لازم داریم؟ (با این روش)

در اینجا باید مستقیم و غیرمستقیم بودن دستور مشخص شود. برای AND مستقیم ۴ کلاک

و برای غیرمستقیم ۸ کلاک لازم داریم. برای کم شدن کلاک ها می توان به طریق زیر عمل کرد:

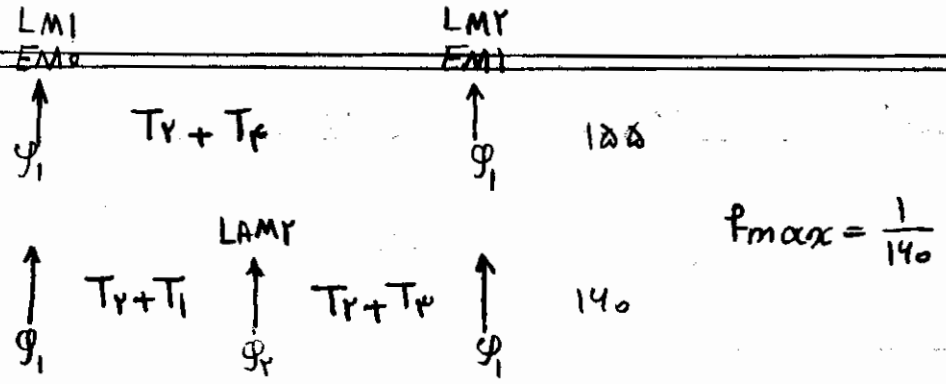
```
      ORG 0
ADD : read  I CALL INDY
      MADD  U JMP Fetch
INDY : DRTAR U JMP Next
      read  U RET
```

(آپرند)

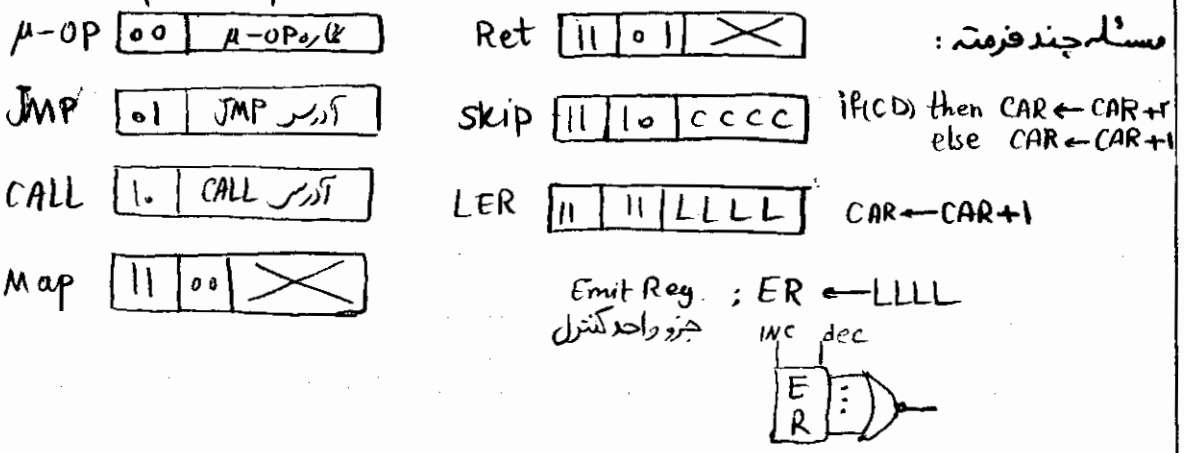
نکته ای که باید به آن توجه داشت این است که در ADD هنگامی که اپرند را می خوانیم با اینکه محتوی

DR و در نتیجه I از بین می رود امّا در همان کلاک I را تست می کنیم ولذا مقدار قبلی I

تست خواهد شد.



تکلیف فصل ۷: ۴, ۵, ۱۳, ۱۴, ۱۵, ۱۶, ۱۷, ۱۸, ۱۹, ۲۰, ۲۲ + مسئله چند فرمته.



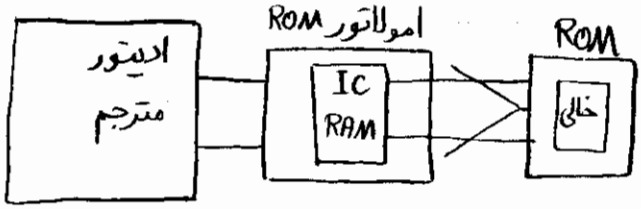
روش میکروپروگرامینگ روشی است از ترکیب نرم افزار و سخت افزار. بعنوان مثال برای ضرب روشهای مقابل با تعداد کلاکهای آن مشخص شده است.

۲۰T	دستور MUL	سخت افزاری
۳۰۰T		نرم افزاری
۸۰ الی ۲۰T	دستور MUL	میکروپروگرام

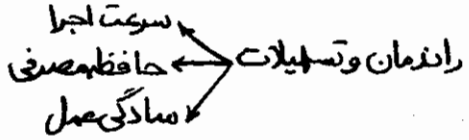
در طراحی واحد کنترل دیدیم که از ROM استفاده می کردیم. در نتیجه هنگام طراحی محتوی ROM را مشخص می کنیم و توسط دستگاه EPROM Programer آن را پر می کنیم. امّا اگر تعداد کلمات ROM بیشتر شود این روش وقت گیر و طولانی خواهد شد. در این حالت از

کامپیوتر و امولاتور ROM استفاده می کنیم . برنامه در کامپیوتر نوشته شده و توسط کامپیوتر

ترجمه شده و وارد امولاتور می‌شود که یک نوع IC RAM است.



فصل ۸ :



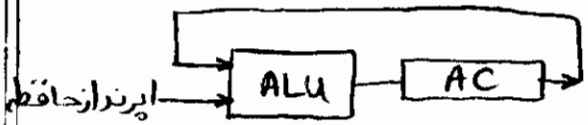
$$\text{CPU}_{\text{سازمان}} = \text{رجیسترها} + \underbrace{\text{ALU} + \text{کنترل}}_{\text{سازمان}}$$

قسمت کنترل را در فصل ۷ دیدیم و بقیه را در این فصل می بینیم.

سازماندهی های مختلف وجود دارند که شامل :

سازمان اکومولاتور ، سازمان جنرال رجیستر ، سازمان RISC و سازمان پشته است.

سازمان اکومولاتور:



در کامپیوترهای اولیه و پروسسورهای کوچک

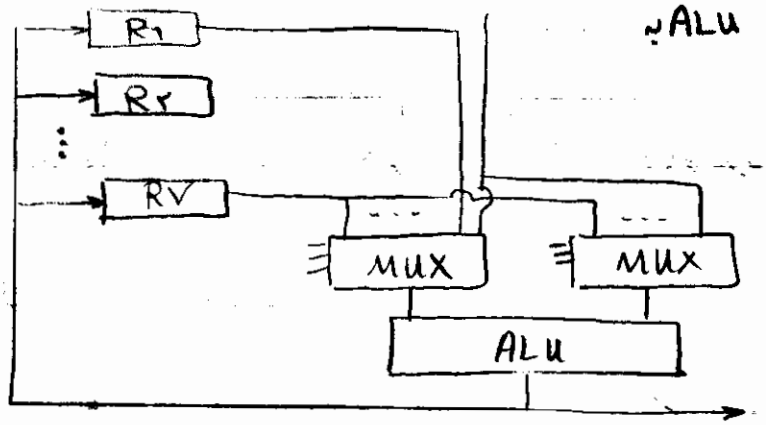
از این سازمان استفاده می‌شود.

سازمان جنرال رجیستر (رجیسترهای عمومی) : در جاهایی که حجم کار وسیع می‌شود و نیاز

به سرعتهای بالا داریم از این سازمان استفاده می کنیم.

به عنوان مثال در دستور ISZ CNT سه بار مراجعه به حافظه داریم (۲ بار برای خواندن از حافظه و یکبار برای نوشتن نتیجه در حافظه) و لذا تعداد کلاک های زیادی نیاز خواهیم داشت در این حالت CNT را به شکل رجیستر پیاده سازی می کنیم و سرعت بالا خواهد رفت. یا مثلاً در مورد بردارها pt (pointer) را نیز در رجیستر قرار می دهیم. این کار حجم مراجعات به حافظه را که وقت گیر است به شدت کاهش می دهد.

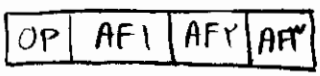
نحوه ارتباط رجیسترها با ALU به شکل مقابل است.



در سازمان آکومولاتور یک AF حافظه وجود دارد. در سازمان جنرال رجیستر ۲ یا ۳ AF حاوی شمارهٔ رجیستر وجود دارد. (آدرس حافظه نوعاً ۱ آدرس است).

سازمان جنرال رجیستر

مثال: ADD R1, R2, R3



نکته ای که مطرح می شود این است که با توجه به اینکه تعداد رجیسترها در مقایسه با تعداد

کلمات حافظه بسیار کمتر است لذا آدرس رجیسترها کوتاهتر از آدرس حافظه خواهد بود.

برای مثال برای ۸ رجیستر ۳ خط آدرس نیاز داریم. کم شدن خطوط آدرس باعث افزایش راندمان خواهد شد. در این سازمان در یک دستور بیش از یک آدرس حافظه استفاده نمی‌شود چون در صورت استفاده طول دستور بسیار طولانی خواهد شد. همچنین هدف در این نوع سازماندهی این است که تمام عملیات را در رجیسترها انجام دهیم.

آدرس رجیسترها — آدرس حافظه

ADD A, R۲, R۳ ، ~~ADD A, B, C~~

در سازماندهی RISC:

پردازش فقط روی رجیسترها است (LD, ST (load, store) با حافظه) و هر دستور ۲ یا ۳ AF حاوی شماره رجیستر است.

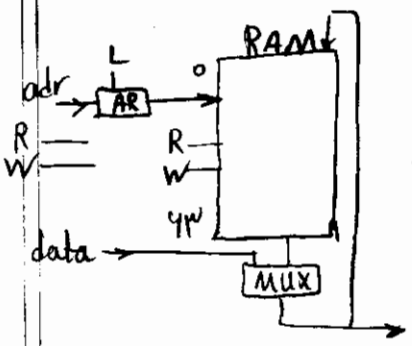
در سازمان پشته: دستور بدون AF است. بدین معنی که اپرندها و آدرس‌ها به طور ضمنی دارای جای مشخصی هستند.

ضبط و بازیابی:

- Random — تصادفی
- stack — پشته — LIFO (last in first out)
- queue — صف — FIFO (first in first out)

می‌خواهیم از یک RAM استفاده کنیم و سه طریق ضبط و بازیابی خوق را پیاده سازی کنیم:

حالت تصاقی را قبلاً دیدیم. برای ضبط سیکل نوشتن را داشتیم و برای بازیابی سیکل خواندن را.



موارد مورد نیاز برای چنین روشی:

ضبط: سیکل نوشتن

T_0: AR ← adr , DR ← data

T_1: M[AR] ← DR

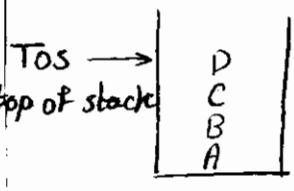
بازیابی: سیکل خواندن

T_2: AR ← adr

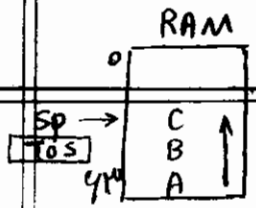
T_4: DR ← M[AR]

برای پشته:

در این حالت ضبط را Push و بازیابی را POP می گوییم. Push یعنی اضافه کردن و Pop یعنی برداشتن از stack.



مفهوم Push بدین معنی است که در هر بار گذاشتن عناصر یک واحد به پائین stack شیفت داده می‌شوند و در Pop هر بار برداشت از stack عناصر یک واحد به بالا شیفت داده می‌شوند بدین ترتیب Tos همواره خانهٔ ثابتی خواهند بود. این روش در مورد RAM قابل استفاده نیست چون در هر Push یا Pop باید تمام عناصر جابجا شود. لذا این گونه عمل می کنیم که Tos را متغیر می گیریم. sp (stack pointer) مکان Tos را نشان می دهد. به این ترتیب که sp اشاره به آخرین جای پر از پشته دارد. همچنین نحوهٔ آدرس بندی



حافظه از بالا به پائین (درخلاف جهت رشد stack) خواهد بود.

با توجه به اینکه یک پشته می تواند سه حالت داشته باشد (پر - خالی - نیمه پر) لذا نیاز به دو بیت داریم که وضعیت پشته را برای ما مشخص کند. (دو بیت E , F) (full empty).

مقدار اولیه : SP ← 0
E ← 1 , F ← 0

ضبط (Push) :

T_0 : SP ← SP − 1 , DR ← data
T_1 : M[SP] ← DR , E ← 0
if (SP = 0) then F ← 1

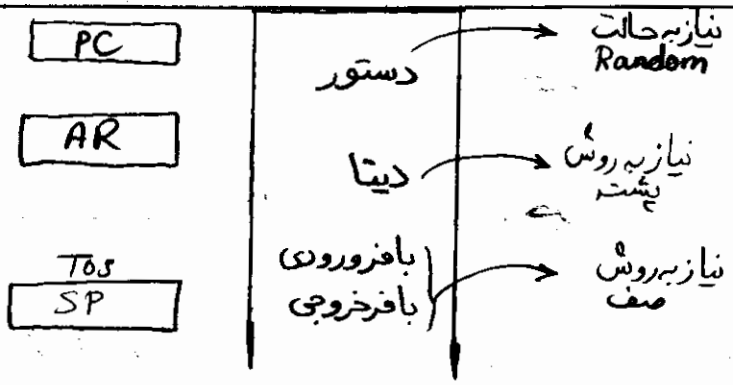
بازیابی (POP) :

T_0 : DR ← M[SP] , SP ← SP + 1 , F ← 0
T_1 : if (SP = 0) then E ← 1

درمورد صف :

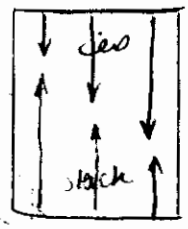
درمورد صف به ضبط ADDQ و به بازیابی DELQ گفته می شود. در این حالت به دو نقطه نیاز داریم. یکی سر صف که محل خروج است و دیگری ته صف که محل ورود است. سر صف و ته صف هیچکدام نمی توانند ثابت باشند. چون همان مشکل ثابت بودن TOS درمورد پشته بوجود می آید. نکته مهم این است که صف به صورت circular (گردشی) است (مانند افرادی که دور یک میز نشسته اند).

ما در یک کامپیوتر به هر سه نوع از این روشها نیاز داریم و هر سه نوع در یک حافظه



پیاده‌سازی می‌شوند.

صف به‌صورت نرم افزاری پیاده‌سازی می‌شود ولی پشته به‌روش سخت افزاری است و رجیستر sp را خواهیم داشت. اضافه‌وکم کردن sp به‌صورت نرم افزاری قابل انجام نیست. stack هایی که درحافظه‌ها پیاده‌سازی می‌شوند نوعاً از آدرس زیاد به کم رشد پیدا می‌کنند (این بهترین حالت ممکن است).



نوعاً در پروسسورها قابلیت load برای sp قرارداده می‌شود. دستورهایی که مرتبط با stack هستند شامل: CALL adr (یک push ضمنی)، RET (یک pop ضمنی)، Ret

pop Ri (مقصد و مبدأ حافظه)، push Ri (مبدأ و مقصد حافظه)

pop all , push all , ...

مثال:

pop AC , push AC , ...

مشاهده می‌کنیم که طول دستور در این حالت کوتاهتر می‌شود چون آدرس حافظه حذف شده است. همچنین وجود دستورهایی چون pushall و popall باعث می‌شود که به یکباره

همهٔ رجیسترها ضبط یا بازیابی بشوند و نیاز به چند دستور نخواهد بود.

به این ترتیب دستورهای push AC و Pop AC جزو دستورهای رجیستری خواهند بود. تاکنون بحث پشته‌ای که داشتیم مربوط به سونوع سازماندهی آکومولاتور و جنرال رجیستر و RISC بود

حال بحث سازمان پشته را شروع می‌کنیم که گفتیم در این سازمان AF نخواهیم داشت.

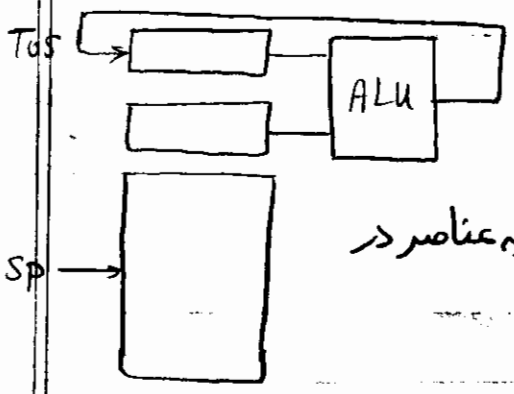
سازمان پشته:

فرم نمایش infix : اپراتور بین اپرندها (A+B)*(C+D)

postfix : اپراتور بعد از اپرندها AB+ CD- * ≡ (A+B)*(C+D)

prefix : اپراتور قبل از اپرندها A BC* +D - ≡ A+(B*C)-D

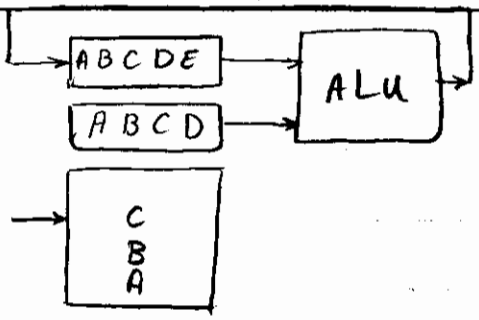
در کامپیوتر با سازمان پشته فرم post fix انتخاب می‌شود. در این سازماندهی به هر اپرند که می‌رسیم آن را به stack ، push می‌کنیم و به هر اپراتور که می‌رسیم دو عنصر بالای پشته را برداشته و اپراتور را به آنها اعمال می‌کنیم و نتیجه را بالای پشته قرار می‌دهیم.



لذا همیشه ALU بر دو عنصر بالای stack عمل خواهد کرد.

دو عنصر بالای stack در دو رجیستر قرار می‌گیرند و بقیه عناصر در حافظه چیده می‌شوند.

مثال: ABCDE + − * /



برای انجام عمل $Z=(A+B)*(C-D)$ در سازمان AC

```
LDA   A
ADD   B
STA   T
LDA   C
SUB   D
MUL   T
STA   Z
```

سازمان AC

آکومولاتور به دستورهای مقابل نیاز داریم:

درحالیکه در سازمان جنرال رجیستر دستورهای مقابل را خواهیم داشت:

```
LD    R1, A
ADD   R1, B, R2
LD    R3, C
SUB   R4, D, R5
MUL   R1, R5, Z
```

سازمان جنرال رجیستر

در سازمان RISC:

```
LD    R1, A
LD    R2, B
ADD   R1, R2, R3
LD    R5, C
LD    R6, D
SUB   R6, R5, R4
MUL   R3, R4, R7
ST    R7, Z
```

سازمان RISC

در سازمان پشته: ←

```
push A
push B
ADD
push C
push D
SUB
MUL
pop Z
```

سازمان پشته

بحث مدهای آدرس دهی:

OP	MOD	AF

مدهای مختلف آدرس دهی را برای رانزمان بهتر و ایجاد تسهیلات استفاده می‌کنیم که شامل:

	AF	EA	OPERAND
۱- ضمنی	نداریم	طبق قرارداد مشخص است.	
۲- بلافصل	در AF خود data را داریم. یا به جای AF خود دیتا را داریم.		
۳- حافظه ای مستقیم	آدرس حافظه	AF	M[AF]
۴- حافظه ای غیرمستقیم	آدرس حافظه	M[AF]	M[M[AF]]
۵- رجیستری مستقیم	آدرس رجیستر	—	محتوای (R)
۶- رجیستری غیرمستقیم	آدرس رجیستر	(R)	M[(R)]
۷- Auto inc	//	//	// , R←R+1
۸- Auto dec	//	//	// , R←R-1
۹- نسبی	یک عدد با علامت	PC+AF	M[PC+AF]
۱۰- index	آدرس حافظه	IX+AF	M[IX+AF]
۱۱- بیس	آدرس حافظه	BR+AF	M[BR+AF]

مراجعات بحافظه

```
LDA   CST   → ۲
STA   CNT   → ۲
LDA   AA    → ۲
STA   pt    → ۲
lop,  ADD   pt I  ۳
      Isz   pt    ۳
      Isz   CNT → ۳
      BUN   lop → ۱
CNT,  Hex 0
CST,  Dec -۸
pt ,  Hex 0
AA ,  Hex 100
```

کاربرد مد بلافصل در ثابت های برنامه است.

به عنوان مثال این نوع کاربرد در برنامهٔ مقابل نشان داده شده است. حال این برنامه را با مد بلافصل می نویسیم.

مراجعات بحافظه

```
      LD   R1, #-۸      ۱
      LD   R۲, #۱۰۰H    ۱
lop,  ADD  (R۲),(R۲), R۳   ۲
*     Isz  R۲           |
      Isz  R1           |
      BUN  lop          |
```

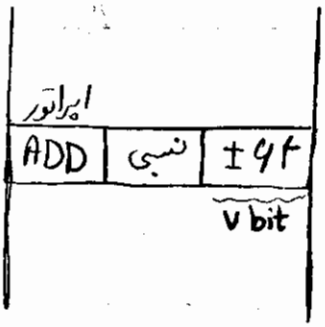
همچنین کاربرد مدهای دیگر نیز آمده است. مشاهده می کنیم که در این حالت طول دستورها کمتر و تعداد مراجعات بحافظه

```
*  lop  ADD  R۳, (R۲)+, R۳
   Isz  R1
   BUN  lop
```

نیز کمتر می شود.

در حالت مقابل مد Auto inc نیز به کار گرفته شده که برنامه کوتاهتر هم شده است.

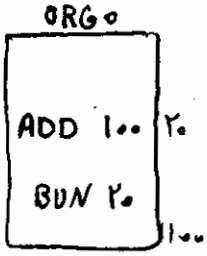
در مد نسبی:



آدرس مورد نظر در یک محدوده ای خاص از دستور قرار می گیرد. در مثال با لا آدرس فوق تا ۶۴ خانه بالاتر یا پائین تری تواند قرار بگیرد. در مد حافظه ای مستقیم اگر دستور

اپراتور	مد	آدرس
ADD را داشته باشیم در مد نسبی	حافظه ای مستقیم	۱۰۰ (۲۰ bit)

می‌بینیم که آدرس ۲۰ بیتی مد حافظه‌ای مستقیم تبدیل به آدرس ۷ بیتی می‌شود که به این ترتیب طول دستور کم می‌شود. پس اولین مزیت کوتاه شدن طول دستور است. مزیت دوم مد نسبی قابلیت جابجایی در حافظه است. به عنوان مثال برنامه مقابل پس از ترجمه حاصل کار باید از خانهٔ صفر به بعد قرار گیرد. اگر بخواهیم برنامه را جابجا کنیم



و مثلاً از آدرس ۱۰۰۰ قرار دهیم دستورات ADD 100 و BUN 20 تغییر می‌کنند. لذا می‌گوئیم، چنین برنامه‌ای قابل جابجایی نیست. مدهای حافظه‌ای مستقیم و غیرمستقیم قابل جابجایی نیستند. مد نسبی قابلیت جابجایی در حافظه را دارد.

مد ایندکس: در حقیقت چنین مدی پیاده‌سازی مفهوم اندیس است.

اپراتور	مد	آدرس شروع بردار
ADD	اندیس	۱۰۰

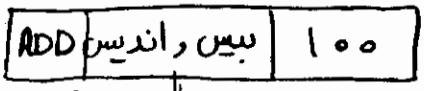
به این ترتیب می‌توانیم به تمام عناصر بردار A رجوع کنیم.

برای اندیس یک رجیستر اندیس وجود خواهد داشت که در آن محتوا نشان می‌دهد به کدام عنصر رجوع می‌شود. (چنین رجیستری مسلماً قابلیت‌های inc, dec را خواهد داشت.)

مد بیس: هدف از این مد مسئلهٔ جابجایی است. اگر بخواهیم برنامهٔ مقابل را به راحتی جابجا کنیم از مد بیس استفاده می‌کنیم.

ADD	حافظه‌ای مستقیم	۲۰۰
ADD	اندیس	۱۰۰
BUN	حافظه‌ای مستقیم	۲۰

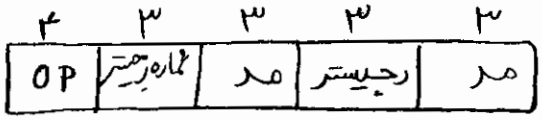
در این صورت مدهد دستور یک عنصر بیس اضافه می‌شود که



محل جابجایی را نشان خواهد داد.

پروسسور PDP یکی از پروسسورهای قدیمی است که سازماندهی آن جنرال رجیستر است.

این پروسسور حدود ۱۰ الی ۱۱ خربت مختلف دستور دارد. یکی از فرمت های آن به شکل

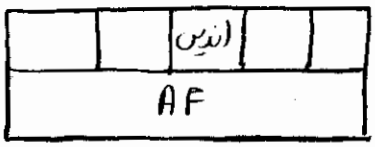


مقابل است:

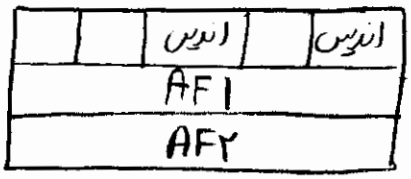
ADD ۰۰۱ ۰۰۰ ۰۱۰ ۰۰۰ ≡ ADD R1, R۲ : R1 ← R1+R۲ (مستقیم)

ADD ۰۰۱ ۰۰۱ ۰۱۰ ۰۰۰ ≡ ADD (R1), R۲ : M[(R1)] ← M[(R1)] + R۲

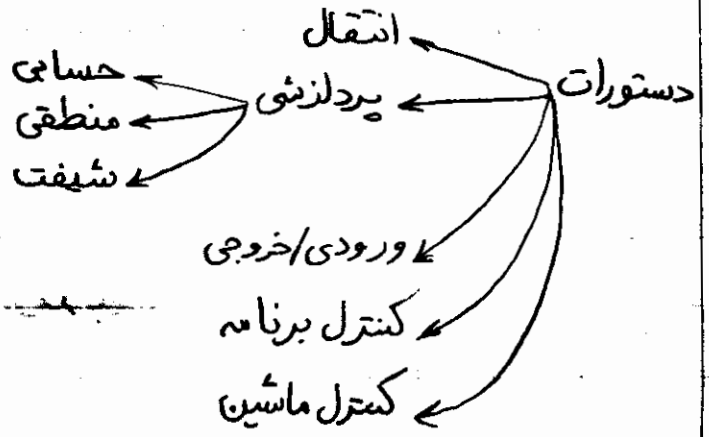
برای مد اندیس می‌توان گفت که اگر هر مدی مد اندیس بود یک کلمهٔ دیگر به دستور اضافه



شود که نشان دهندهٔ آدرس (AF) خواهد بود.



جهت بعدی راجع به تنوع op-code خواهد بود:



از دستورهای انتقال موارد مقابل را داریم : LD , ST , MOV , XCH , push , pop
(load, store, move, exchange; انتقال بین رجیسترها، جابجایی دو رجیستر)

دستورهای بالا هر کدام با مدهای مختلفی به کار می روند که تنوع زیادی ایجاد می کند.

دستورهای پردازشی : ADD , SUB , MUL , DIV , ADC , SBB , inc, dec
(ADC: Add with carry)

این دستورات نیز با مدهای مختلف به کار می روند. علاوه بر این تنوع ها در پروسسورهای مختلف

هر دستور دارای فرم های مختلف است مثلا : (نوع اپرند متفاوت)
(binary)
ADD bin
ADD BCD
ADD FL
ADDDP
⋮

مثلاً وقتی حجم داده ها زیاد ولی پردازش بر روی آنها ساده است

از ADDBCD استفاده می کنیم.

دستورات منطقی : CMP (Complement), AND, OR, XOR, CLA, CLE, SETE (set بیت ها)

دستورات شیفت : cir, cil, shr, shl, Ashr, Ashl, cirtc (to carry), cilte

فرم کلی یک دستور شیفت

۴	۱	۲		
شیفت	L/R	نوع	اپرند	تعداد شیفت

OP code

در مقابل نشان داده شده است که در پروسسورهای مختلف به کار می رود.

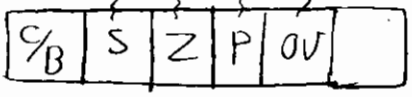
دستورات ورودی/خروجی : INP Padr , Out Padr (port address)

با توجه به اینکه یک پروسسور می تواند چندین دستگاه ورودی و خروجی داشته باشد لذا باید

					CD	برای اعداد با علامت		برای اعداد بدون علامت	
Ret	call	adr	JMP	adr	1	JGT	adr	JHT	adr
			JC	adr	C	JGE		JHE	adr
			JNC	adr	C′	JLT		JlowT	adr
			JZ	adr	Z	JLE		JlowE	adr
			JNZ	adr	Z′	JEQ		JEQ	adr
			JP	adr	S′	JNE		JNE	adr
			JN	adr	S				
			JPO	adr	P				
			JPE	adr	P′				
			JOV	adr	OV				
RNOV			CNOV	adr	JNOV adr	OV′			

program status word

PSW:



	C/B	S	Z	P	OV
A+B	۱	۰	۱	۰	۰
A−B	۰	۰	۰	۱	۱
A∧B	—	۰	۰	۱	—

A ۱۰۰۱

B ۰۱۱۱

بیت C/B فقط در جمع و تفریق بدون علامت معنی دارد و در با علامت استفاده ای از آن نمی‌شود. بیت OV (overflow) در جمع و تفریق با علامت معنی دارد.

بیت C/B در جمع بدون علامت معنی سرریز دارد و در تفریق رقم قرضی محسوب می‌شود (یعنی $A<B$ بوده است). لذا در جمع بدون علامت برای فهم سرریز باید به بیت C/B نگاه کنیم نه بیت OV. در زبانهای سطح بالا علامت اعداد مشخص می‌شوند ولی در زبان اسمبلی خودمان باید مشخص کنیم و دستور مناسب با آن را استفاده کنیم.

برای تشخیص رابطه بین B,A (از نظر بزرگتر و کوچکتر بودن) می‌توانیم تفریق آنها را تشکیل

دهیم و سپس بیت ها را تست می‌کنیم. دستورهایی که در صفحهٔ قبل برای اعداد با علامت و بی‌علامت نشان داده شده اند همراه با دستور Comp A,B می آیند.

به عنوان مثال :

```
Comp A,B
JGT adr
```

Comp A,B تفریق A و B را تشکیل می دهد ولی حاصل تفریق را نگه نمی دارد بلکه ارزش Flag ها را تعیین می‌کند. دستور Comp جزو دستورات کنترلی است. دستور دیگر مشابه دستور Comp، دستور TEST است که به صورت TEST A, Mask به کار می رود. این دستور A و mask را با هم AND می‌کند و مانند Comp حاصل AND را نگه نمی دارد.

$CD(JGT) = OV'.S'.Z' + OV.S$

$CD(JHT)^{?} = (CB)'Z'$ Jump Higher than

وقفه ها	حداقل :	مسائل وقفه
سیگنال وقفه	$M[SP] \leftarrow PC$	ضبط وضعیت
سیکل وقفه	$SP \leftarrow SP - 1$	تشخیص عامل
	$PC \leftarrow$ Const / M / Bus	تعیین اولویت
روتین وقفه	$IEN \leftarrow 0$	رفتن به روتین با اولویت و اجرا
		بازیابی وضعیت برگشت

در سازماندهی های جنرال رجیستر ... برای ضبط رجیسترها در stack به شکل

مقابل عمل می‌کنیم:

```
push psw
push R1
push R2
 :
```

و برای بازیابی به شکل روبرو:

```
 :
pop R2
pop R1
pop psw
ION
RET
```

البته از دستوراتی مانند push all , pop all استفاده می‌شود.

در پروسسورهای پیشرفته در سیکل وقفه کار مقابل نیز

M[SP] ← PSW
SP ← SP-1

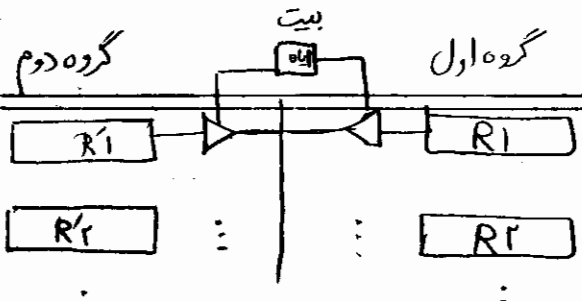
انجام می‌شود:

و در نتیجه در مسائل وقفه دستور RETI (RET from Interrupt) اضافه می‌شود که دقیقاً برعکس سیکل وقفه است.

$$RETI \begin{cases} POP\ PSW \\ ION \\ RET \end{cases}$$

در مورد $PC \leftarrow \begin{cases} Const \\ M \end{cases}$ روش pulling (نرم‌افزاری) انجام می‌شود که برای تشخیص و تعیین اولویت است. در مورد $PC \leftarrow BUS$ از روش Vect INT (سخت‌افزاری) استفاده می‌شود که در آن تشخیص و تعیین اولویت رفتن به روتین سخت افزاری در سیکل وقفه انجام می‌شود.

در بعضی از پروسسورها دیگر عمل ضبط انجام نمی‌شود بلکه دو گروه رجیستر وجود دارد. در حالت عادی گروه اول و در وقفه گروه دوم رجیسترها استفاده می‌شود.



وقفه
- سخت افزاری (عامل یک سیگنال است)
  - خارجی
  - داخلی
- نرم افزاری.

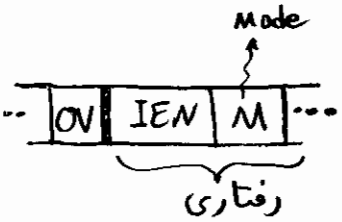
در وقفه‌ٔ سخت افزاری خارجی ، وقفه توسط یک دستگاه خارجی انجام می‌شود. در داخلی عامل وقفه یک سیگنال است و وقفه توسط مدارهای داخلی که خودمان طراحی کرده ایم ایجاد می‌شود. این مدارها برای تشخیص خطاها طراحی می‌شوند.

در وقفه نرم افزاری ، وقفه توسط یک دستور انجام می‌شود.

یک نوع call خاص: INT ۱۰ / INT ۲۰ — برای رجوع به سیستم عامل

call فوق با call معمولی که می‌شناسیم تفاوت دارد. call فوق برای رجوع به سیستم عامل برای دریافت یک سرویس خاص انجام می‌شود. سرویس خاص می‌تواند خواندن یک سکتور از دیسک باشد. نوعاً سیستم عامل به تمام چیزهایی دسترسی دارد که دیگران به آن دسترسی ندارند. (دیگران: مثلاً User).

در PSW یک قسمت رفتاری وجود دارد :



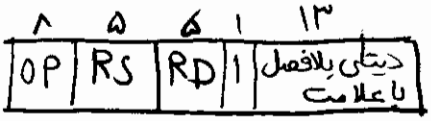
۰ یا ۱ بودن بیت Mode محدودهٔ کار را برای User یا سیستم عامل مشخص می کند.

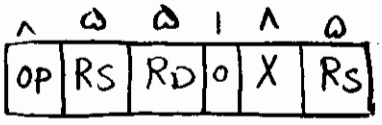
هنگامی که به سیستم عامل رجوع می کنیم بیت Mode عوض می شود و رجیسترها در اختیار سیستم عامل قرار می گیرند و User دیگر نمی تواند کاری بکند. بعد از خاتمهٔ کار سیستم عامل دوباره بیت Mode عوض می شود و به برنامهٔ User برمی گردیم.

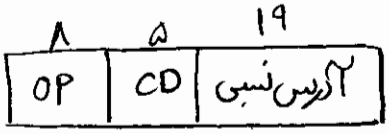
کامپیوترهای

RISC	CISC
دستورات کم و موثر	دستورات زیاد
مدها کم و موثر	مدها زیاد
فرمت ها کم و با طول ثابت	فرمت دستورات متعدد و با طرحهای مختلف
آدرس ها در پردازشی فقط در رجیستر	آدرس اپرند در پردازشهای تواند باشد.
کنترل سخت افزاری	کنترل میکروپروگرام است.
رجیسترهای زیاد با پنجره های overlap	
حافظهٔ دستور و دیتا جدا.	
اجرای دستور به طور متوسط در یک کلاک.	

مثال:







برای ADD ⟵ ADD R۱, R۲, R۳

برای MOV ⟵ ADD R۱, R۰, R۲ R۰ [۰]

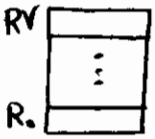
برای INC ⟶ ADD R1, #1, R1

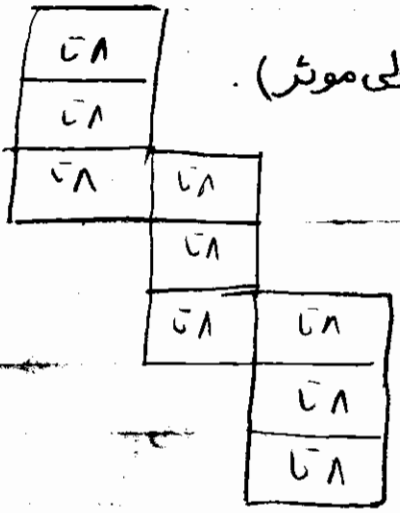
برای Dec ⟶ ADD R1, #-1, R1

می‌بینیم که یک دستور ADD برای چندین کار استفاده می‌شود و بسیار موثر است.

(نشان دهندهٔ دستورات کم ولی موثر).

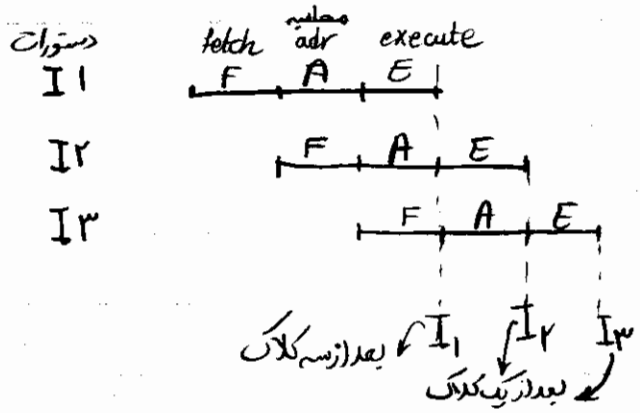
رجیسترها با پنجره‌های





با overlap :

اجرای هر دستور به طور متوسط در یک کلاک توسط پردازش موازی Pipeline انجام می‌گیرد.



میکروپروسسورها:

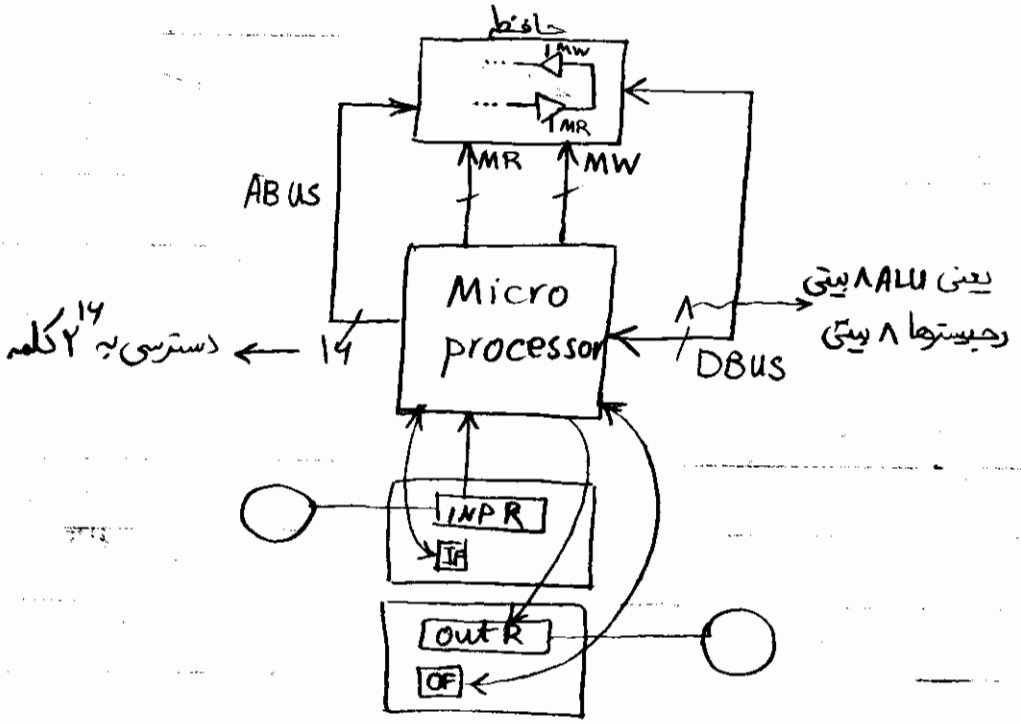
دیدیم که cpu شامل رجیسترها، ALU و واحد کنترل است. اگر کل cpu در یک IC قرار بگیرد به آن میکروپروسسور می گوئیم.

اگر cpu و قسمتی از حافظه و قسمتی از مدار واسط در یک IC باشد به آن Single micro Computer

گفته می شود که برای کاربردهای کوچک و خاص استفاده می شود. البته این امکان وجود دارد که حافظه ها و مدارهای واسط دیگری در خارج داشته باشیم.

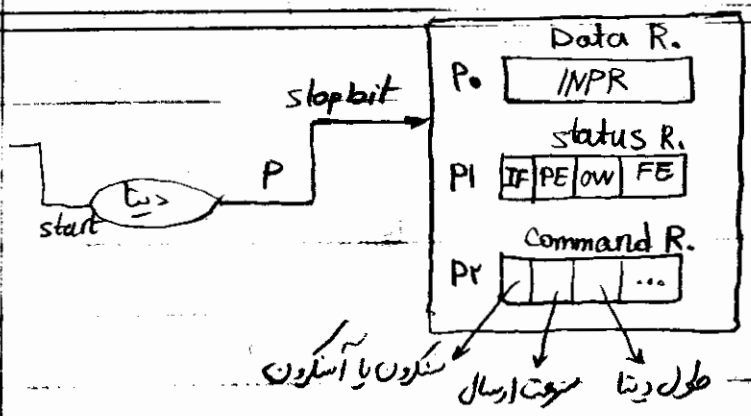
اگر علاوه بر موارد بالا، A/D، تایمر، کانتر و... نیز در یک IC قرار بگیرند به آن میکروکنترلر می گوئیم.

یک میکروپروسسور شامل CBUS (Control)، DBUS (Data)، ABUS (Adr)



عموماً مدارهای واسط به شکل بالاست بلکه از سه گروه رجیستر تشکیل می شود:

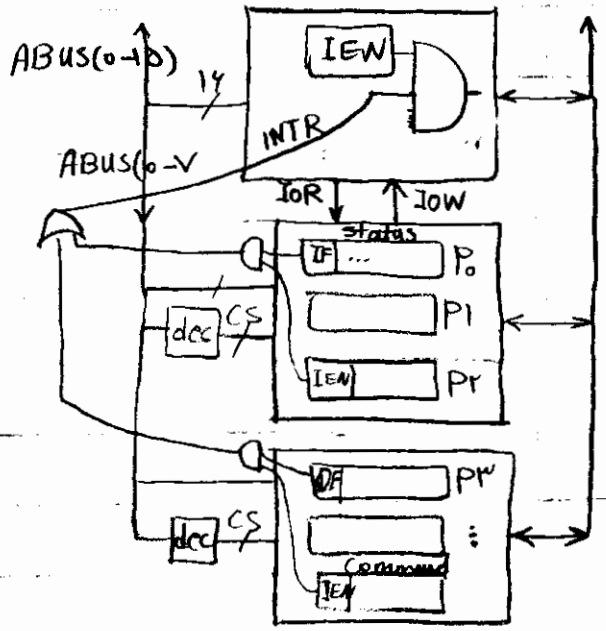
مدار واسط ورودی



PE : parity error

OW : overwrite

به متعلقات مدار واسط (سه رجیستر بالا) port گفته می‌شود. (P_0, P_1, P_2)



: program I/O

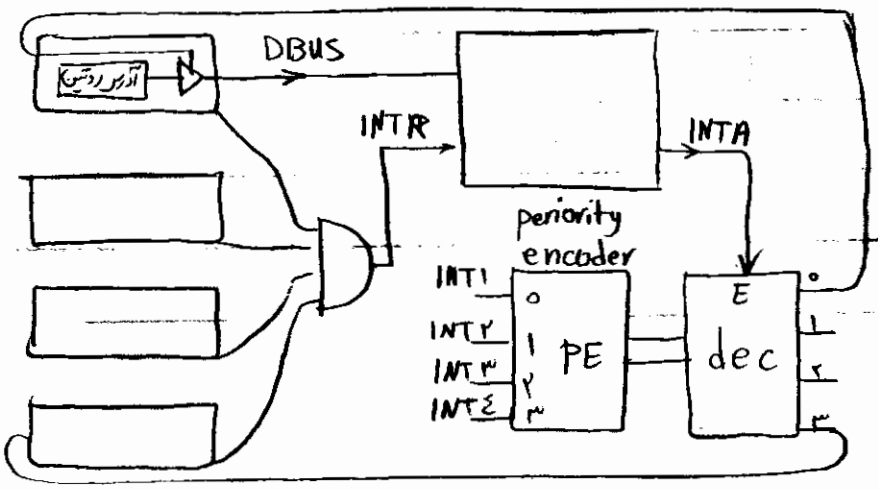
```
LI, SKI
    BUN LI
    INP
    ⋮
```

```
LI, INP P0
    Test 1000...0
    JZ LI
    INP P1
    ⋮
```

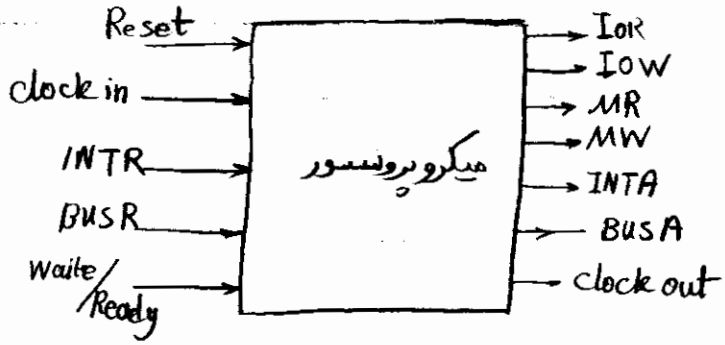
vector interrupt

: Vec INT

اولویت‌بندی افزاری دقیق



در حالت کلی :


Reset
clock in
INTR
BUSR
Waite/Ready
میکروپروسسور
IOR
IOW
MR
MW
INTA
BUSA
clock out