منبع: معماری کامپیوتر موریس مانو (انتشارات ...) دکتر قدرت سپیدنام جدیدترین ویرایش

۳ فصل اول درس داده نمی‌شود.

۶ و ۷ اصلی است.

۱۴ نمره پایان ترم - ۶ نمره کوییز کلاسی (با تکلیف کوییزها)

۱٫۴ حضور و غیاب

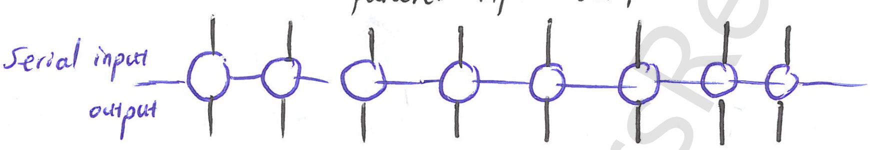
+

۲۱٫۴ نمره

hsayyari.blog.ir

فصل ۴: رجیسترها: رجیسترها (Registers) حافظه‌های کوچک با سرعت پردازش بالا در cpu هستند! هر Register تعدادی مشخص از فلیپ‌فلاپ‌ها است که کنار هم قرار می‌گیرند. (هر فلیپ‌فلاپ 1 بیت را در خود نگاه می‌دارد)

شیفت Register یونیورسال:



هم شیفت register است و هم رجیستر موازی و سری نیز ورودی و خروجی دارد!

Register ها را در معماری به این گونه نشان می‌دهیم: R_1

ریز دستورالعمل: هر دستوری که منجر به تغییر محتویات یک Register شود. (Micro operation)

* مهم‌ترین تفاوت cpu ها در ساختار operation ها است.

زبان RTL: (Register transfer language) زبان انتقال ثبات: همین operation ها را با RTL نمایش می‌دهیم:

دستور انتقال در RTL: $\overset{5}{R_1} \leftarrow \overset{7}{R_2}$ (ریختن محتویات R_2 در R_1)

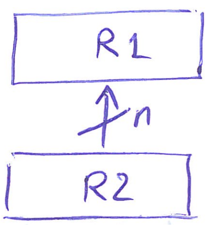
* در انتقال move انجام نمی‌شود کپی انجام می‌شود!

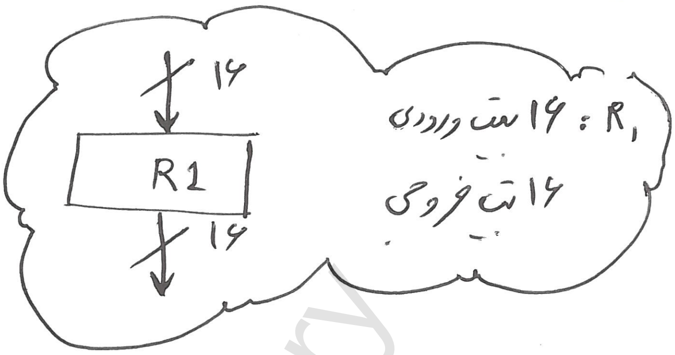
مثال از شرط : C++ : در : if (x>10) y=4;

در معماری Q : $R_1 \leftarrow R_2$

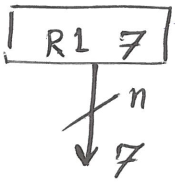
Q ← مداری کنترلی است

مثال عملکرد داده :

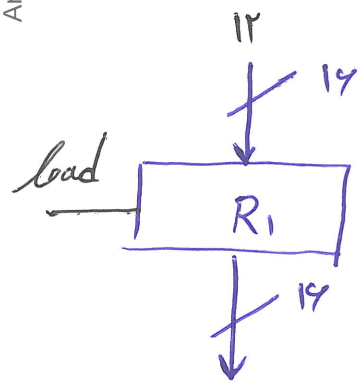




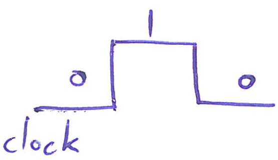
** هر چه در Register است در خروجی آن نیز هست! یعنی هیچ چیز در خود نگه نمی‌دارد!

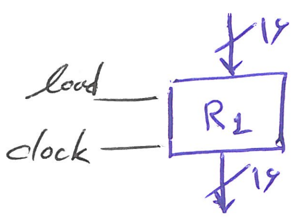


* هر Register یک پایه Load دارد که برای ورود اطلاعات به Register باید فعال باشد! یعنی برای خروجی نیاز به اجازه نیست ولی برای ورود اطلاعات به رجیستر نیاز به اجازه هست!



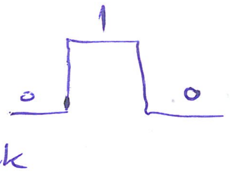
* برای ورود اطلاعات علاوه بر Load نیاز به clock نیز می‌باشد که برای کنترل بهتر در مدار آن را حساس به لبه می‌گیریم و رفتار مدار را کنترل می‌کنیم.

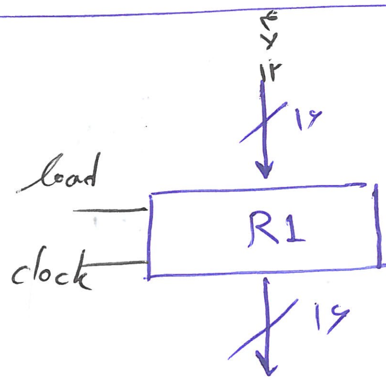




صفر ۳ دارد را!

{ دارای puls یعنی حساس به سطح }

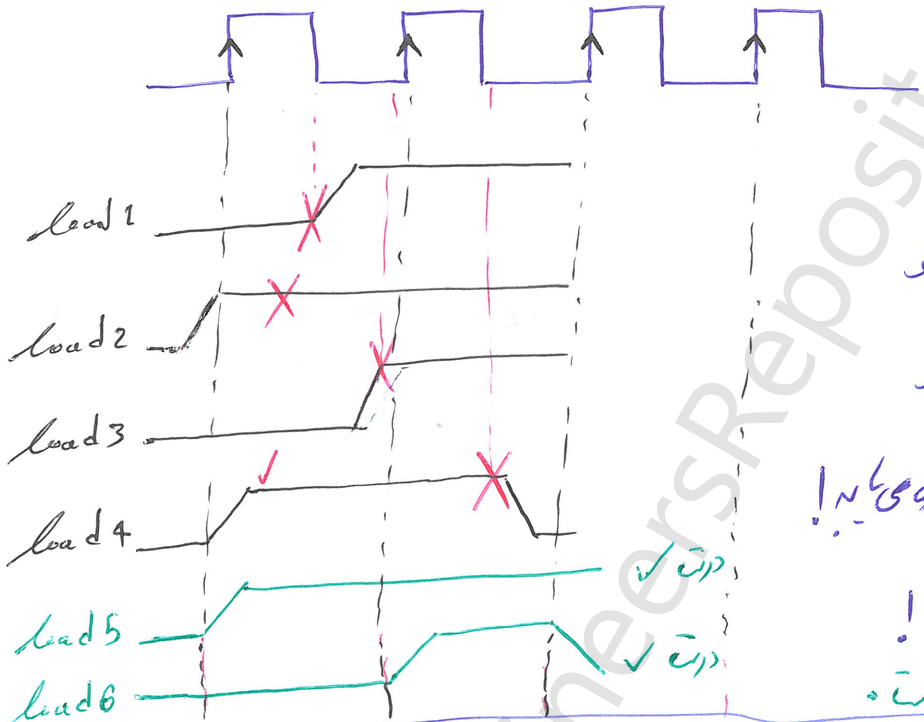




*. اگر clock سطح باشه فقط ۴ را می‌فرستیم و ۱۲ و ۷ را نه!

*. اگر load لبه باشه اجازه ورود اطلاعات نمی‌دهد

*. clock حساس به لبه و load حساس به سطح است.



*. طرح ۴ load بالا غلط است چون تغییر فقط در pulse کلاک اجازه می‌یابد!

نکته: این لبه بالا رونده است!

/// دوتای پایین درست است.

دستور swap در C++ :

x_1, x_2

$y = x_1$

$x_1 = x_2$

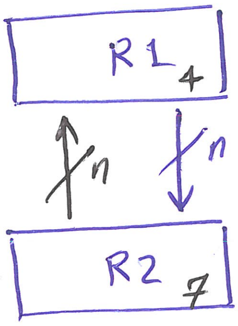
$x_2 = y$

دستور swap در معماری :

اگر T برقرار بود $T: R1 \leftarrow R2$ و $R2 \leftarrow R1$

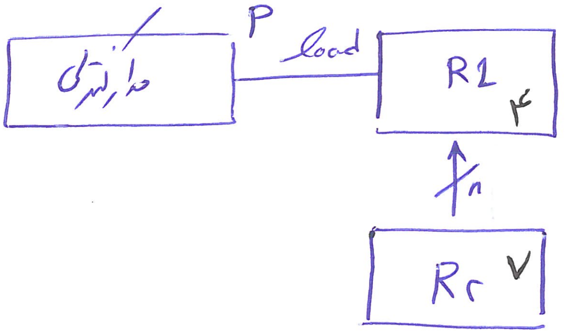
این دو دستور

$TQ: R1 \leftarrow R2$ این جا دو شرط T و Q، and شده‌اند یعنی هر دو باید فعال باشند!

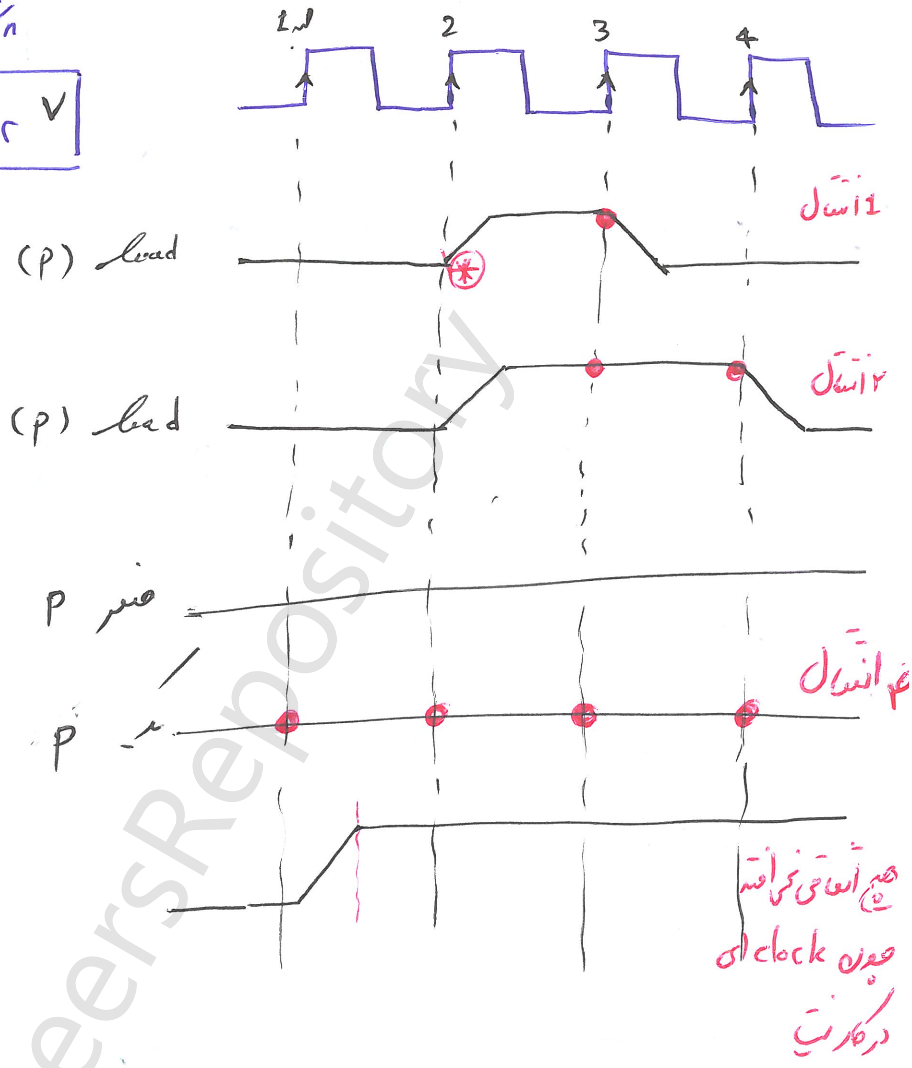


*. پس حالا ۴ شرط نیاز داریم: load R1 فعال / load R2 فعال / کلاک روی لبه باشد / T هم برقرار باشد!

$T+Q: R1 \leftarrow R2$ T و Q هم or شده‌اند

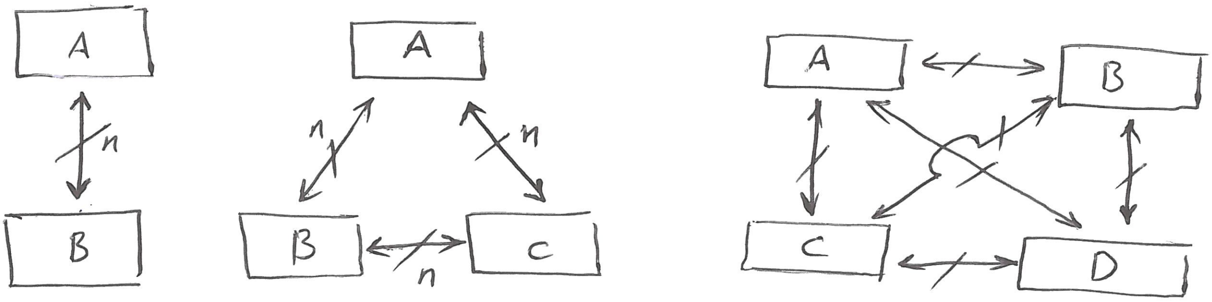


$P: R_1 \leftarrow R_2$ دستور انجام دهید



(*) در اینجا انتقال نمی افتد چون گفتیم که load باید حداقل یک سطح ولتاژ بوده باشد پس انتقال در لبه ۳ صورت می گیرد.

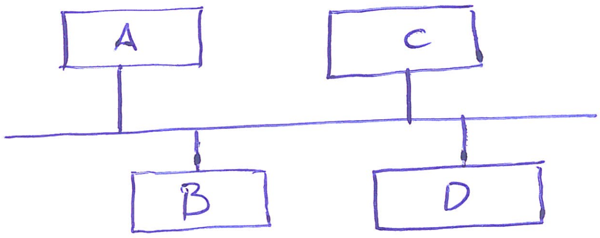
Bus :



شکل فوق :
(هر اتصال هایی برای ارتباط بین رجیسترها)

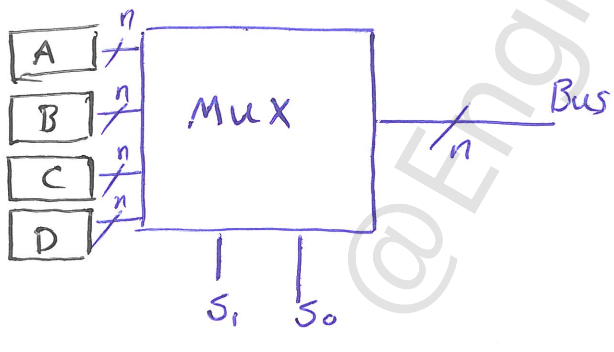
* شکل فوق باوجود سرعت بالای اتصال دارای پیچیدگی ارتباط بین این رجیسترها است.

پس برای حل مشکل ارتباط پیچیده، سیستم Bus طراحی گردید:
(مزیت این روش سادگی طراحی و امکان استفاده از تعداد زیادی رجیستر)



در این روش هرچه تعداد رجیسترها بیشتر شود، راندمان سرعت کمتر می‌شود.

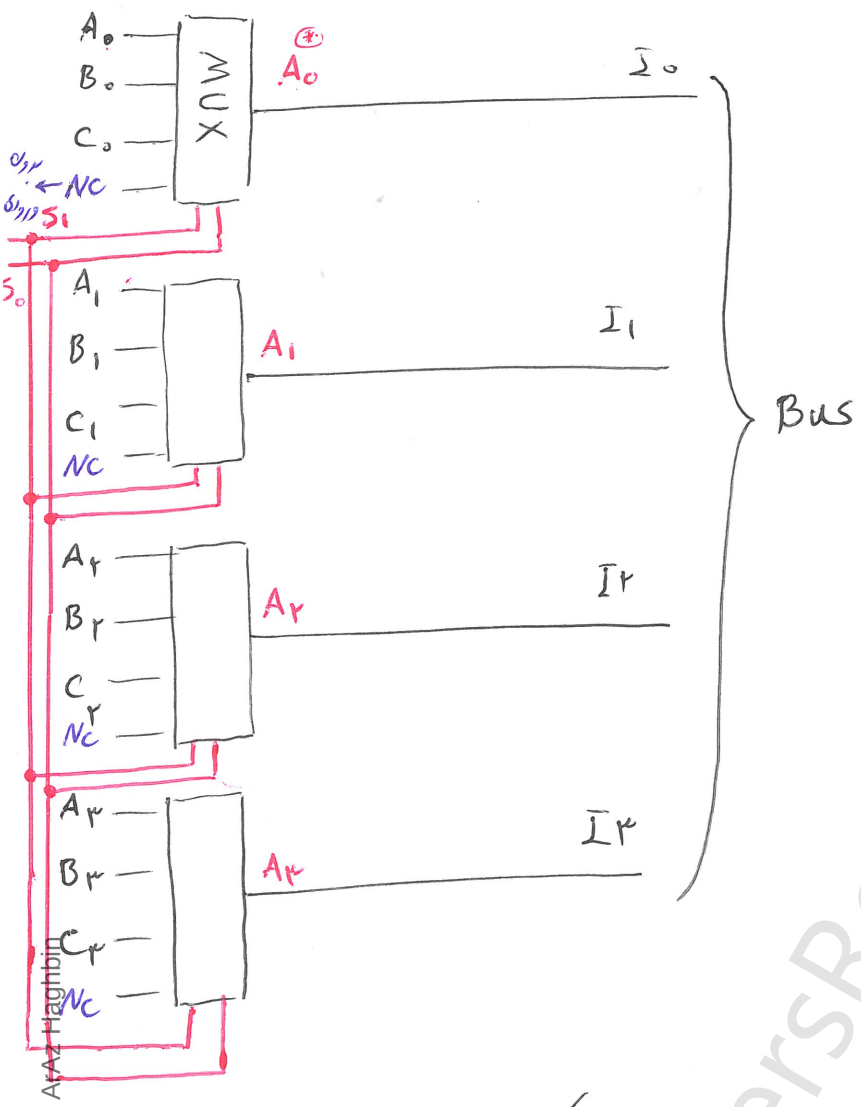
* نکته: چون هر چیزی در درون رجیسترها است در خروجی آن هم است پس باید مکانیزمی برای ایجاد ترتیب در سیستم Bus ایجاد کرد. برای این کار از Multiplexer استفاده می‌کنیم: (شکل مقابل)



مثال: یک گذرگاه مشترک برای ۳ رجیستر ۴ بیتی طراحی کنید:

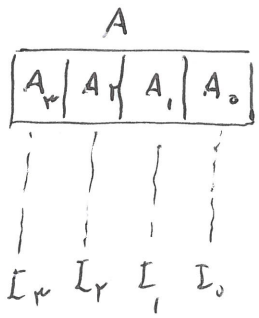
نکته: تعداد خطوط Bus معمولاً برابر تعداد بیت‌های رجیسترها است!

ادامه صفحه بعد!

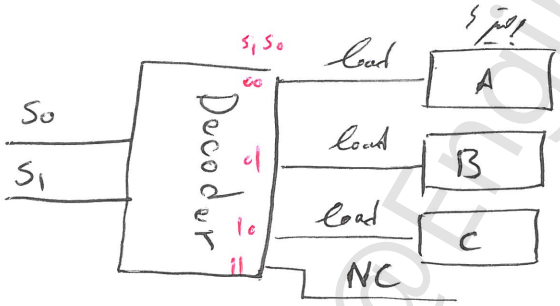


S_0 و S_1: خطوط ترکی (انتخاب) است که با ۰ و ۱ دادن به آن‌ها روی کدام ورودی (رجیستر) روی Bus می‌رود را انتخاب می‌کنیم. مانند A که نشان دادیم!

برای مثال ورودی A:



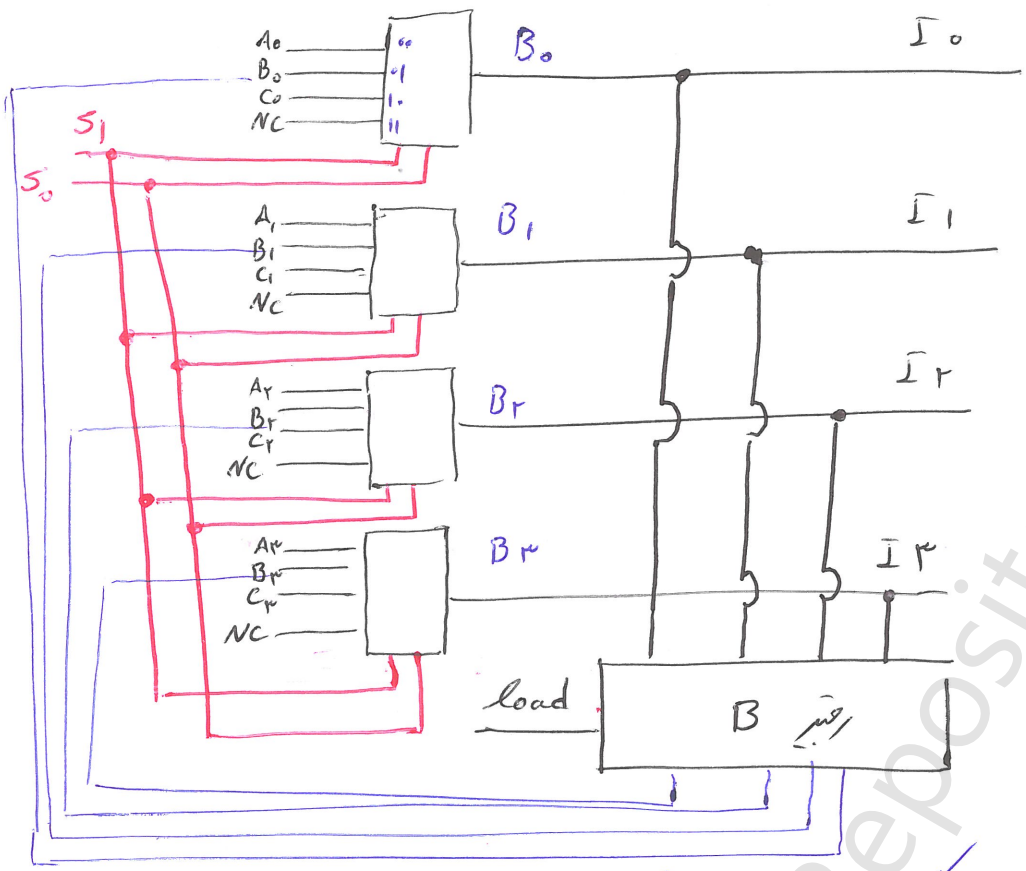
برای اینکه کدام یک از رجیستر‌ها بازنویسی (load) شود از مدار decoder استفاده می‌کنیم که ورودی load رجیستر‌ها را به خروجی وصل می‌کند:

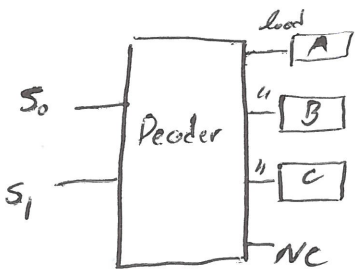


موفق و پیروز باشید!

صفحه ۳ دارد!

روش اول راه‌اندازی از روی Bus و MUX:



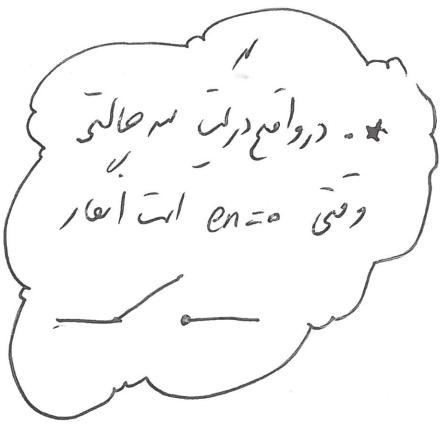


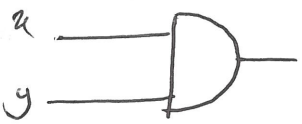
* دریک ۳ حالته از enabler فعال یا نه کنترل (مدار) کار می‌کند:

ورودی	خروجی
0	0
1	1

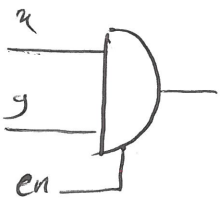
en	ورودی	خروجی
0	X	X
1	0	0
1	1	1

* در واقع در یک سه حالتی وقتی en=0 است آن مدار



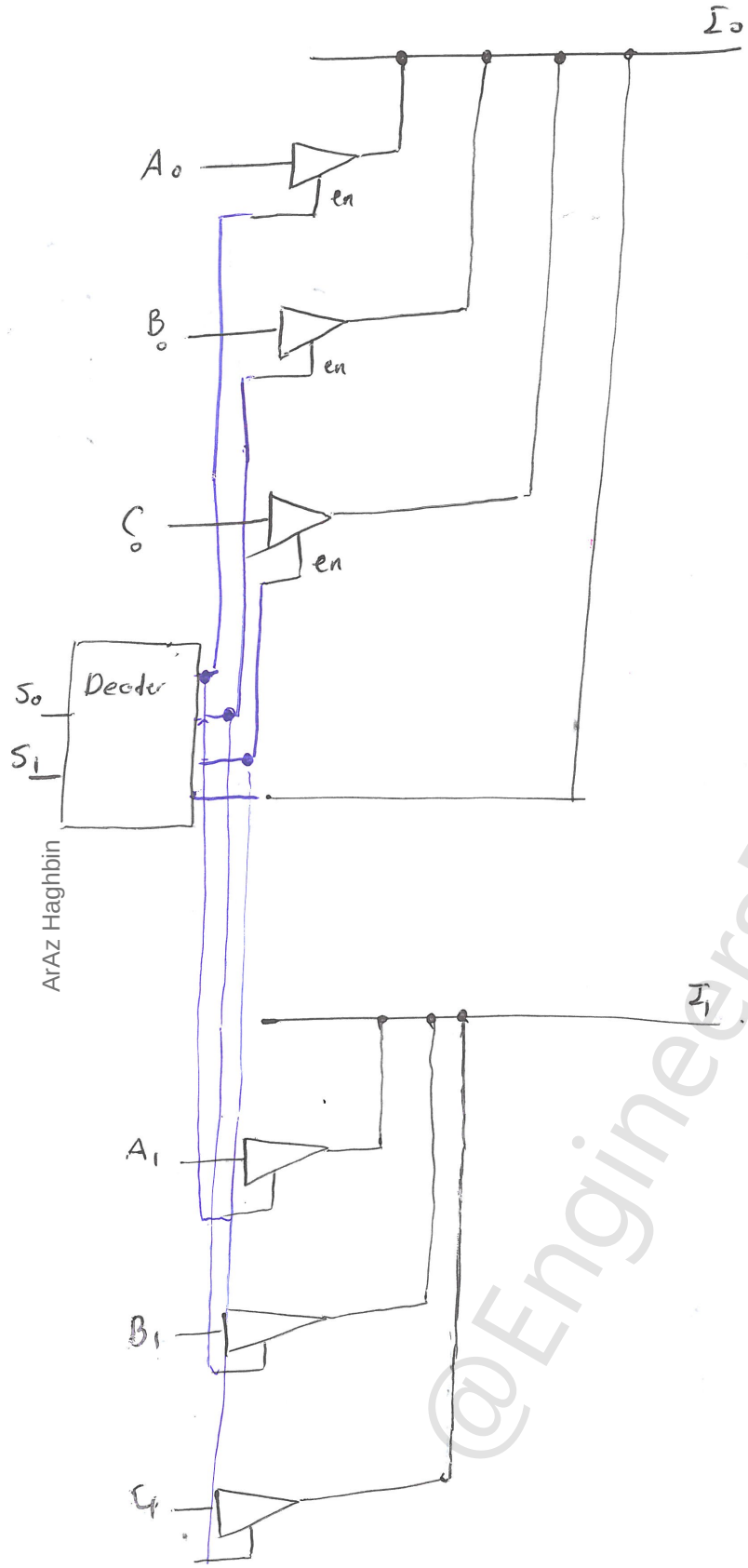


x	y	خروجی
0	0	0
0	1	0
1	0	0
1	1	1



en	x	y	خروجی
0	X	X	X
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

روش دوم برای برداشتن از روی Bus بالت ۳ حالته:

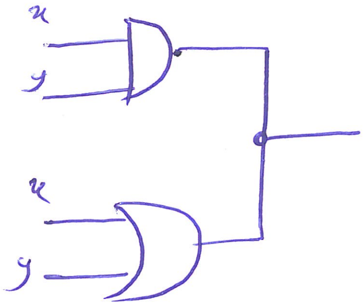


این قطعه کد خطا است! (I_0)

کد decoder را هم خودتان کامل کنید!

ادامه دارد...

صفحه 5 دارد!



* شکل روبه رو فقط در دو شکل امکان دارد درست باشد!

۱- گیت ۳ حالته باشد

۲- و در هر لحظه فقط یک گیت فعال باشد!

$R_2 \leftarrow R_1$

$R1 \xleftarrow{Read} M[AR]$

$M[AR] \xleftarrow{write} R1$

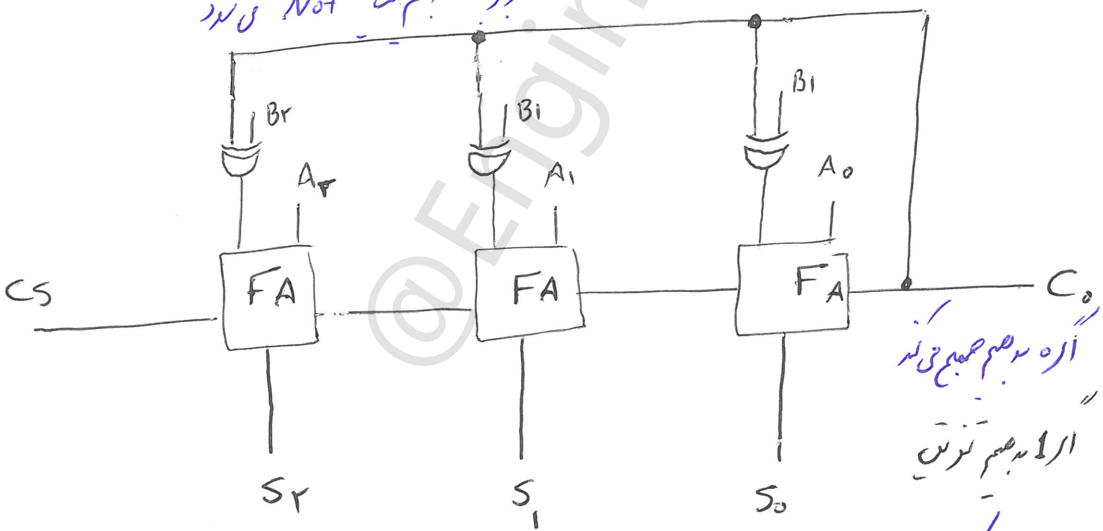
$A - B = A + \overline{B} + 1$

نکته در مورد گیت Xor :

x برابر ورودی گیت Buffer می‌شود

x برابر ۱ ورودی گیت Not می‌شود

x	y	خروجی
0	0	0
0	1	1
1	0	1
1	1	0

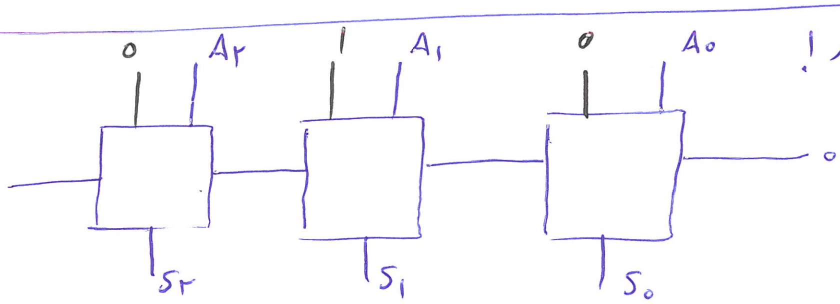


C_0 ⟶ نقش selector مدار را بازی می‌کند!

اگر ۰ بدهیم جمع می‌کند

اگر 1 بدهیم تفریق

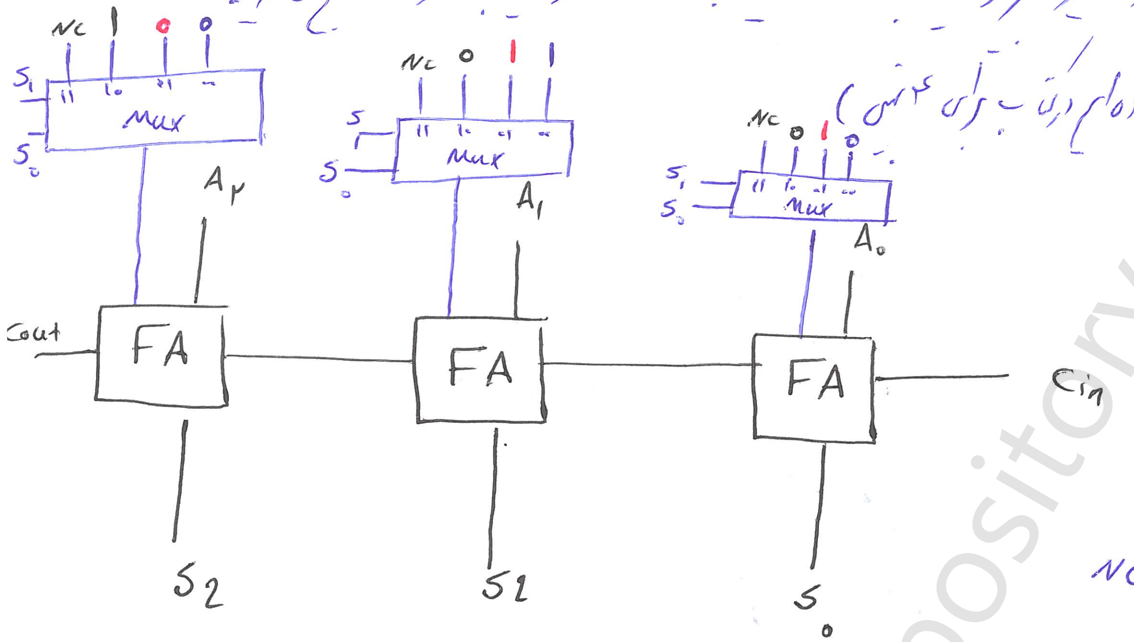
مداری که هر چی می‌دهیم (۳ بیتی) به علاوه ۲ کند!



تکمیل صفحه A ثابت:

مداری طراحی کنید که مقدار رجیستر A را با عدد ۲ یا با عدد ۳ و یا با عدد ۴ جمع کند.

(از Mux برای ۳ تای اعداد انجام داده و یک بیت برای عملیات)



$2 \rightarrow 010$

$3 \rightarrow 011$

$4 \rightarrow 100$

NC: No connection!

با Mux توانستیم چهار کار را با FA انجام دهیم و در این مدار سه کار را انجام داده‌ایم.

carry ورودی را صفر در نظر بگیریم

S_1	S_0	خروجی
0	0	A+2
0	1	A+3
1	0	A+4
1	1	X

carry ورودی ممکن است ۱ باشد

S_1	S_0	C_{in}	خروجی
0	0	0	A+2+0
0	0	1	A+2+1 (3)
0	1	0	A+3+0
0	1	1	A+3+1 (4)
1	0	0	A+4+0
1	0	1	A+4+1
1	1	0	X
1	1	1	X

$A - B = A + \overline{B} + 1$

A_3	A_2	A_1	A_0	
1	1	1	1	= A−1

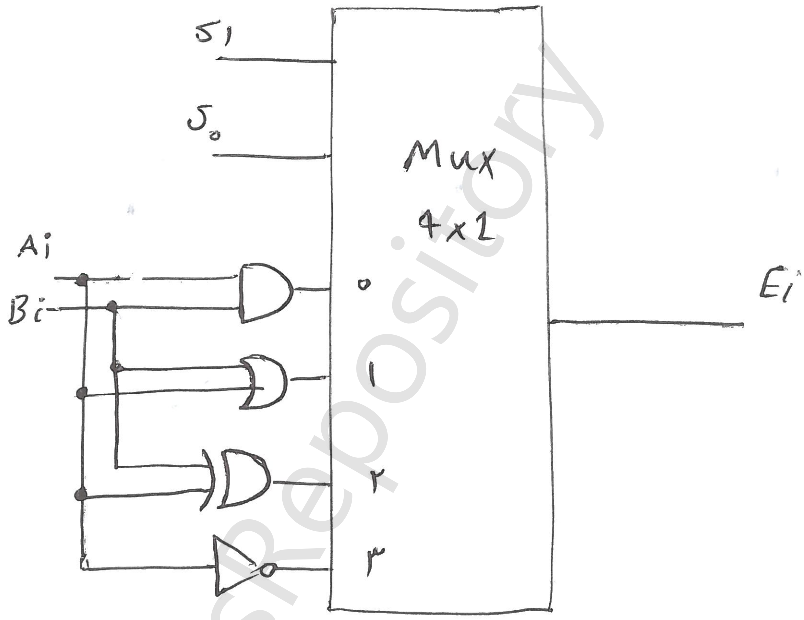
متمم ۲ یک

چون دارای ۲ متمم ۱ عدد را جمع می‌کنیم

* وقتی عددی (A) را با متمم ۲ عددی دیگر (B) جمع کنیم در واقع داریم A−B را انجام می‌دهیم.

جدول صفر ۱۱۰ الف ب حل شود.

* یک مدار از Mux بخش منطق ALU را انجام می‌دهد. این کار برای هر بیت واحد شده بی
اگر Register 4 بیتی بود 4 Mux نیاز داشتیم. Mux در Selector که مشترک باشد.



Selective Set :

مرحله ⓪ می‌خواهیم این دو بیت 1 شود

	A_3	A_2	A_1	A_0
A =		1	1	
B =	0	1	1	0

① از رجیستر کمکی داریم

② دو بیت نظیر را در رجیستر B برابر 1 قرار می‌دهیم

OR →

مرحله ③ این دو را با هم OR می‌کنیم که باید 1 شده باشند

Selective Complement :

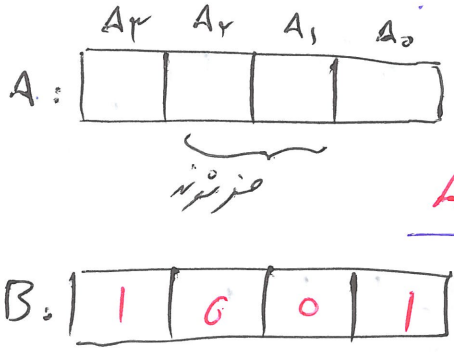
	A_3	A_2	A_1	A_0
اصلی A =				
		متمم	متمم	
کمکی B =	0	1	1	0

XOR →

صفر ۳ را دارد!

		XOR
0	0	0
0	1	1
1	0	1
1	1	0

صفر کردن بیت‌های دلخواه از A (MASK): Selective clear



Insert: به جای بیت‌های خواهیم در بخشی از رجیستر بگذاریم. مقدار مطلوب:

A = 0 1 0 0 1 1 0 1

1 0 1 0

1 1 0 0 0 0 1 1

(1)-

And: 0 1 0 0 0 0 0 1

OR: 0 0 1 0 1 0 0 0

0 1 1 0 1 0 0 1

مثال:

A = 1 0 1 1 0 1 1 0

11 01

And: 0 0 1 1 1 0 0 1

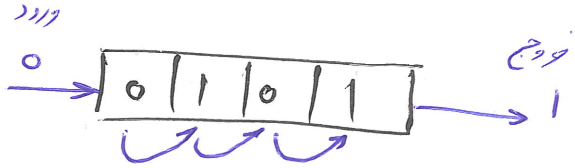
0 0 1 1 0 0 0 0

OR: 1 1 0 0 0 0 1 0

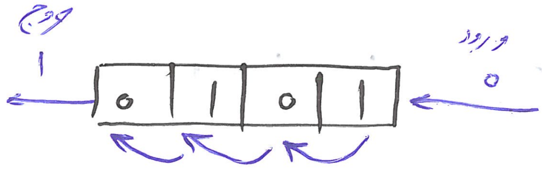
1 1 1 1 0 0 1 0

انواع shift : shift حسابی
shift منطقی
shift چرخشی

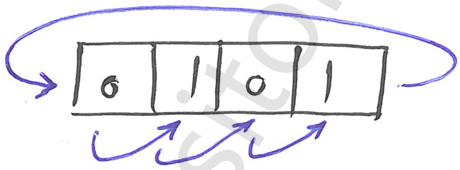
شیفت منطقی به راست :



شیفت منطقی به چپ :

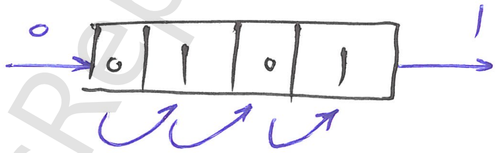


شیفت چرخشی به راست :



در شیفت حسابی عددی که در رجیسترها است عدد علامت‌دار است.

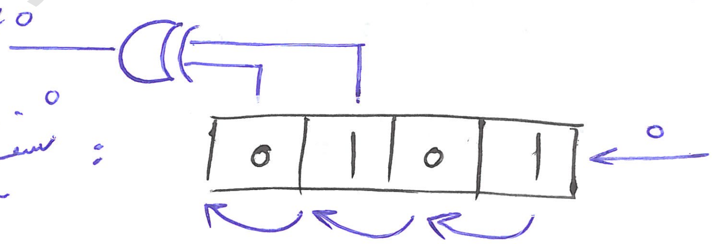
حسابی به راست :



0 یعنی مثبت
1 یعنی منفی

* رجیستری که عدد علامتدار دارد شیفت حسابی می دهند و اگر نه شیفت (shift) منطقی.

شیفت حسابی به چپ :

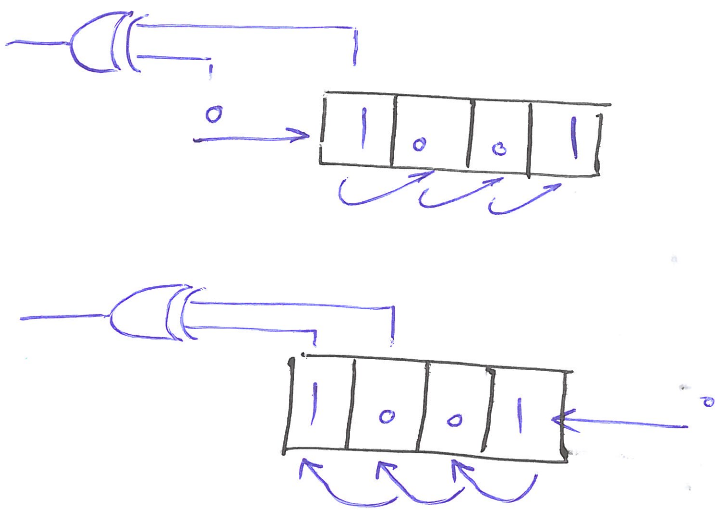


* اگر دو بیت آخر مخالف العلامه بودند اجازه shift نداریم. (در حسابی)
به همین علت از Xor استفاده می کنیم.

هر شیفت به چپ ۲x (ضرب در ۲) و هر شیفت به راست باید ۲/ (تقسیم بر ۲) شود

شکل ۴-۱۲ : بیت 0 از بیرون وارد می شود!

ALU:
- ص ۱۰۸ قسمت حسابی
- ص ۱۱۱ منطقی (قسمت منطق)
- شیفت منطق



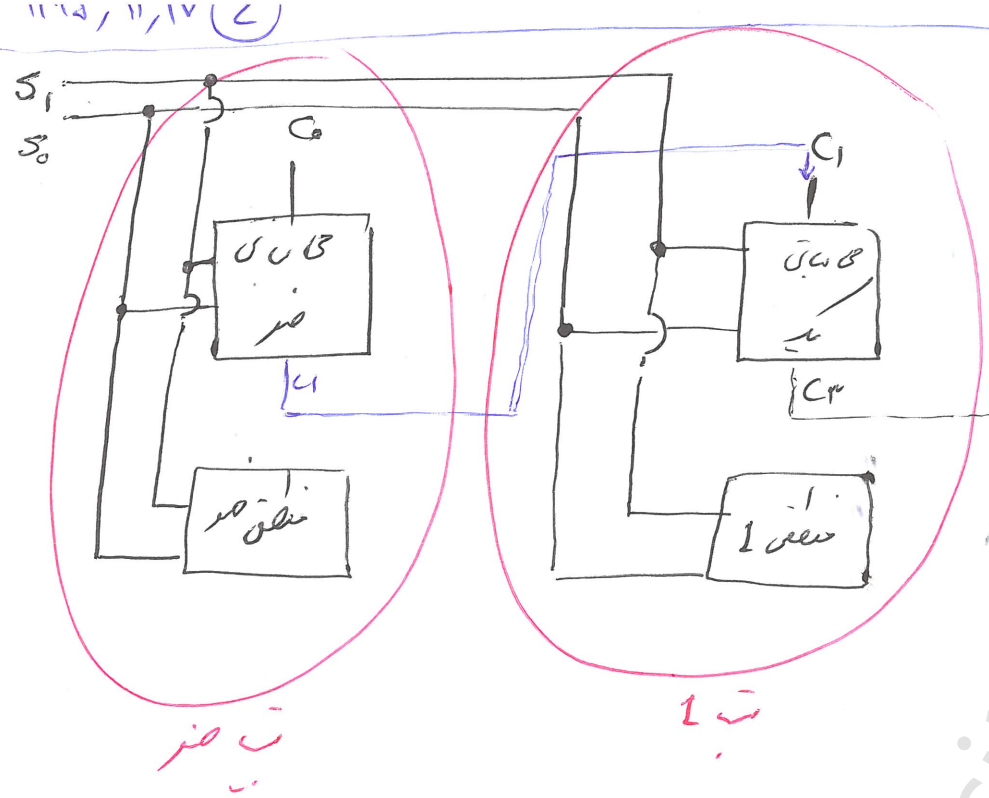
در شکل صفحه ۱۱۷ داریم:

در اینجا کدام حالت ها گرفته:

S_3	S_2	خروجی
0	0	مدار حسابی
0	1	مدار منطق
1	0	شیفت به راست
1	1	شیفت به چپ

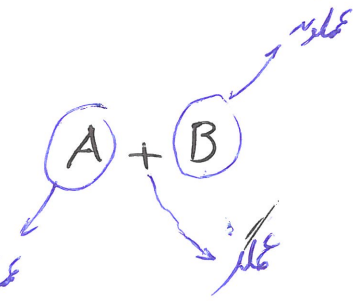
	S_3	S_2	S_1	S_0	C
A+B	0	0	0	0	0
A+1	0	0	1	0	1
AB	0	1	0	0	X
A⊕B	0	1	0	0	X
shiftLA	1	1	X	X	X

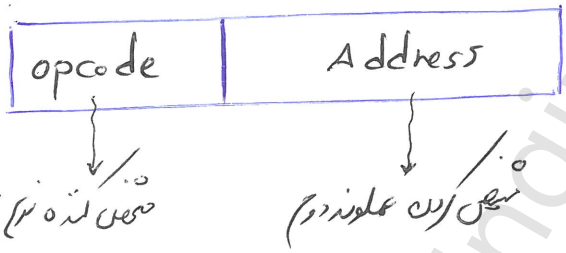
* بالا صفحه ۱۱۷: به تعداد ALU است مثلاً اگر ۶۴ بیت ۶۴ طبقه داریم که خطوط انتخاب ($S_0, S_1, \dots$) و carry آن از طریق به هم وصل است.



فصل ۵ :

مالک دستورالعمل :





* AC : Accumolator.
رجیستری که بیشتر کارها از قبیل محاسبات و عملوند اول و جواب در آن ریخته می‌شود.

زبان RTL → $-R \uparrow$ → می‌شود → $AC \leftarrow AC - R1$

* این که opcode چند بیتی باشد بستگی به تعداد دستورالعمل‌های دارد
(مثال: برای ۲۰ عمل نیاز به ۵ بیت داریم)

* Address : این که چند خانه را باید تعداد حافظه انتخاب می‌کند
(مثال: برای ۱۶ خانه ۴ بیتی خواهیم $16 = 2^4$ ✓)
حجم ۳ دارد! نیاز به ۵۰ خانه ۶ بیتی خواهیم $64 = 2^6$ ✓)

مثال: قالب دستورالعمل برای کامپیوتر طراحی کنید که دارای ۴۰ دستور باشد و از یک حافظه ۱۵۰۰ خانه‌ای که هر خانه ۶۴ بیت باشد استفاده کند:

۶ بیت	۱۱ بیت
opcode	Address

ارتباطی با طراحی قالب دستورالعمل ندارد!

* این کامپیوتر احتمالاً توی رجیستر ۶۴ بیتی دارد (می‌تواند کمتر باشد ولی دوبار می‌خواند اگر بیشتر باشد هم که ...)

روشهای آدرس‌دهی:

add (جمع کردن نه آدرس)

بلافصل (Immediate):

Add	25

$Ac \leftarrow Ac + 25$

Direct:

Add	25

$Ac \leftarrow Ac + M[25]$ Ac را با خانه 25 حافظه جمع کن

Indirect:

Add	25

$Ac \leftarrow Ac + M[M[25]]$

برو خانه ۲۵ آدرس جای دیگر را بده تا با آن جمع کنی را بردار بیار

* در حالت Direct و Indirect را می‌توان با یک بیت از هم جدا کرد در خود دستورالعمل است: (منظور)

ساختار:

1	3	12
	opcode	Address

Addressing mode →
- 0 Direct
- 1 Indirect

① روش یک (با کم کردن یک بیت از opcode):

مثال:

دستور اولیه:

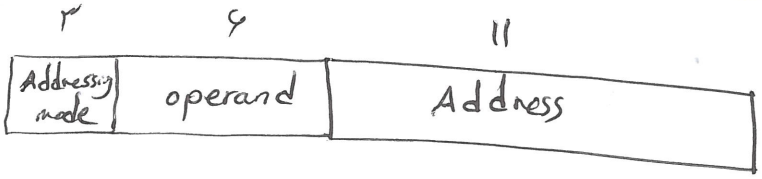
۴	۱۲

دستور با Addressing mode:

۱	۳	۱۲

مثال فرضی: (۱۶ Addressing mode)

۳ بیت برای Addressing mode نیاز داریم



* طول دستورات ۱۶ بیتی است

ولی در واقعیت نباید

رجیستر را ۳۲ بیتی طراحی کنیم تا به صورت 2^{32} باشد!

آدرس حافظه: آدرس واقعی عملوند! شکل صفحه ۱۲۸ کتاب

DR: Data Register: هر عملیاتی نیاز به ۲ عملوند دارد که یکی AC و دیگری حتما DR است.

AC: محل ورود برای (رجیستر محل ورود) عملیات ۲ عملوندی.

PC: (programm Counter) محل آدرس دستورالعمل بعدی را در خود نگه می‌دارد.

AR: Address Register: هر دستور حافظه توسط رجیستر AR صورت می‌گیرد.

IR: Instruction Register: رجیستر خاصی که هر دستوری که از حافظه واکشی می‌گردد داخل آن قرار می‌گیرد (مثال صفحه ۱۲۸ شکل ب)

10	ADD	457

در IR قرار می‌گیرد).

در IR تنها دستورات قرار می‌گیرد (نه مقدار)

TR: Temp Register: رجیستر کمکی که در محاسبات میانی کاربرد دارد. (مقدار، نه دستور)

INPR: Input Register: هر داده‌ای که می‌خواهد از بیرون از CPU وارد کامپیوتر شود اول وارد INPR می‌شود.

OUTR: Out put Register: هر داده‌ای که می‌خواهد از CPU خارج شود، اول وارد OUTR می‌شود.

شکل صفحه ۱۳۱ به کتاب مراجعه شود.

* چون ۳ selector داریم می‌توانیم حداکثر ۸ و حداقل ۵ انتخاب انجام دهیم.
* حالت (۰ ۰ ۰) تعریف نشده است.
* چیزی از حافظه روی Bus بیاید: ۱- selector ، ۷ انتخاب شود (۱۱۱)

↓ حالت پایین:

۱- آدرس انتخاب شده روی AR
۲- فعال کردن پایه خواندن
۳- انتخاب selector ۷ (۱۱۱)

نوشتن روی حافظه از Bus:

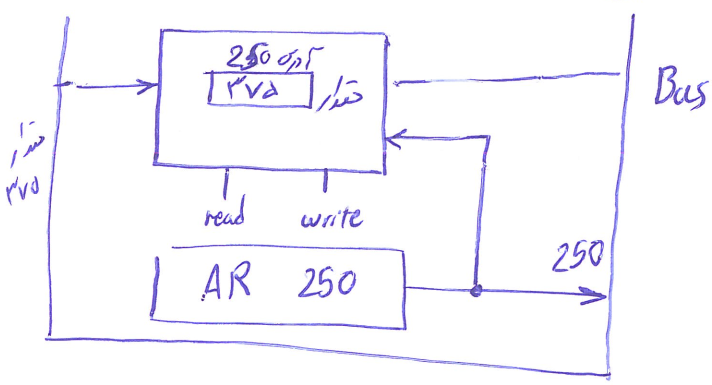
۱- آدرس روی AR
۲- فعال کردن پایه نوشتن (load)

* نکته: load حالت نوشتن است. به selector نیاز داریم چون selector برای آوردن روی Bus است نه برداشتن از آن.

INR: پایه Increment که مقداری داخل رجیستر است را اضافه 1 می‌کند.

مثال: خانه ۲۵۰ حافظه را روی Bus بگذاریم:

۱- آدرس خانه ۲۵۰ روی AR حافظه



* مقدار مثلاً از DR
آدرس از AR

شکل صفحه ۱۳۱:

$AR \leftarrow PC$: ۱- selector (۰۱۰) 2

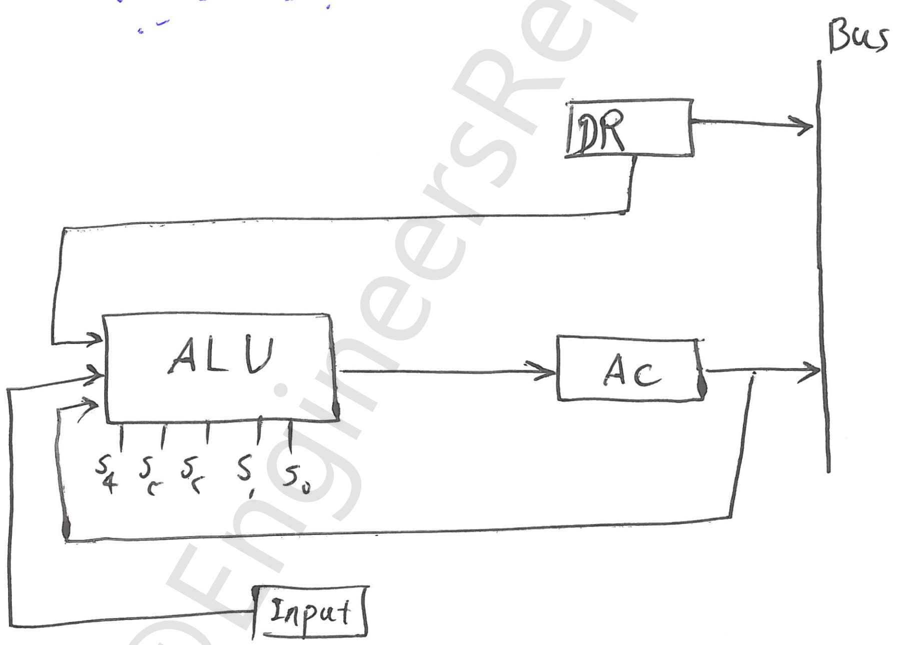
۲- فعال کردن load و AR

* چیزی از Bus وارد AC نمی‌شود

$AC \leftarrow AC + DR$: ۱- رجیستری را از قصد در Bus نمی‌گذاریم)

۲- فقط باید selector و ALU عملیات را مشخص کنیم

۳- load فعال AC فعال شود تا حاصل عملیات به آن انتقال یابد.



* چرا که می‌خواهیم در AC قرار دهیم اول باید در DR قرار دهیم بعد به AC می‌رسیم:

$PC \leftarrow PC + AR$; (PC → AC، AR → DR)

۱- selector (Bus) را ۲ (۰۱۰) می‌گذاریم PC وارد Bus شود
۲- load و DR را فعال می‌کنیم تا PC وارد DR شود.
۳- selector های ALU را طوری تنظیم می‌کنیم DR را رد کند
۴- load و AC را فعال می‌کنیم تا PC وارد AC شود.

صفحه ۳ دارد!

۵- selector را 1 (001) تنظیم می‌کنیم تا AR وارد Bus شود.

۶- load DR را اتصال می‌کنیم تا AR (از روی Bus) روی DR قرار گیرد.

۷- selector های ALU را به گونه‌ای تنظیم می‌کنیم که DR را به علاوه AC کند (DR+AC)

۸- load AC را اتصال می‌کنیم تا از ALU وارد AC شود

۹- selector را به گونه‌ای تنظیم می‌کنیم که مقدار AC را روی Bus قرار گیرد (4) (100)

۱۰- load به PC را اتصال می‌کنیم تا اطلاعات از روی Bus وارد PC شود.

---

* IR تنها به پایه load دارد چون فقط یک دستور در آن وجود دارد (پس نیازی به +1 شدن یا clear کردن ندارد). IR کار محاسباتی ندارد.

* در دیگر رجیسترها این صفر کردن clear (محاسبات) به کار می‌رود.

Input ⟵ output

۱- تنظیم selector ALU تا Input را رد کند

۲- load AC را اتصال می‌کنیم

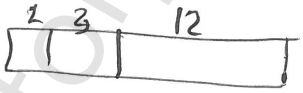
۳- Bus selector (4) 100 را اعمال می‌کنیم تا روی Bus بیاید

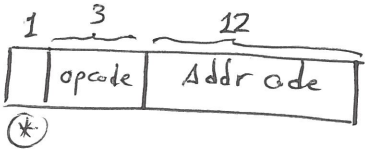
۴- load outpu را باز می‌کنیم!

$TR \leftarrow TR + DR$ سوال امتحانی! (مهم)

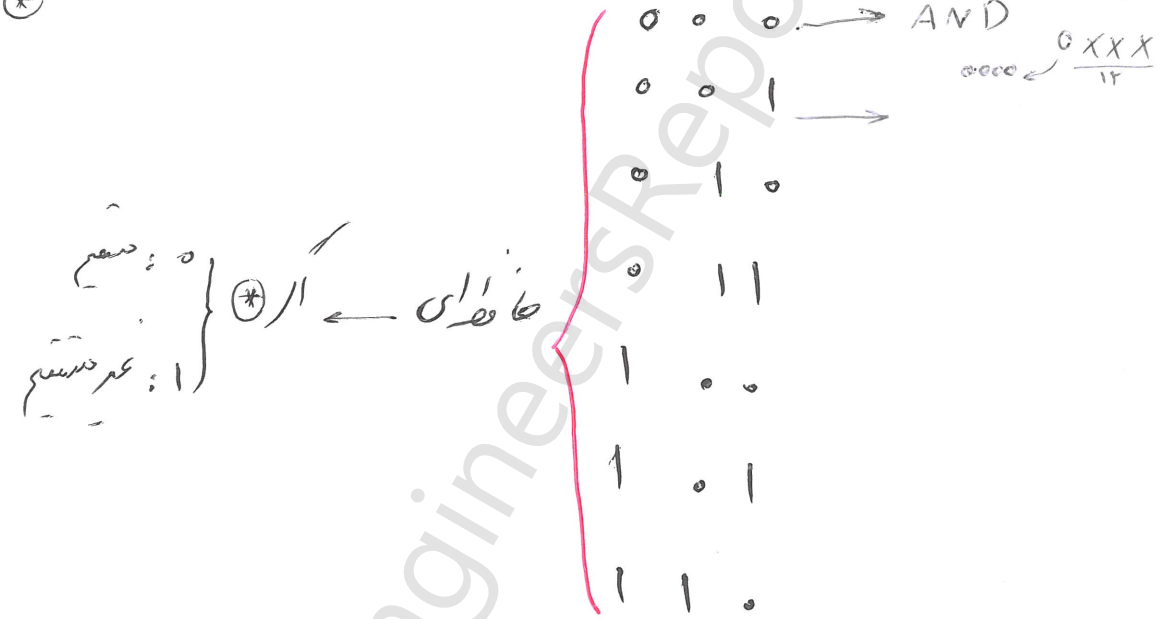
۳ نوع دستور:
- حافظه‌ای: انجام عملیات با داده‌ای از حافظه (محتویات خانه ۳۷۵ را با AC جمع کن)
- رجیستری: دستورات رجیستری (DR را با PC جمع کن و نتیجه را در AC)
- ورودی/خروجی (I/O): رجیستری ورودی و خروجی (از CPU)

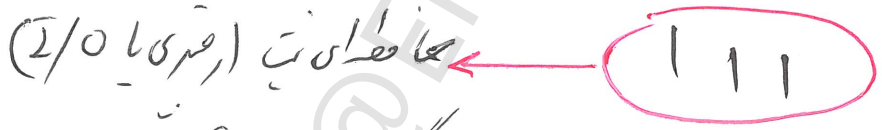
* دستورالعمل در رجیستر IR قرار می‌گیرد!
PC آدرس دستورالعمل بعدی را نگه می‌دارد.
* تحلیل decode کردن می‌گوید! (از راست)

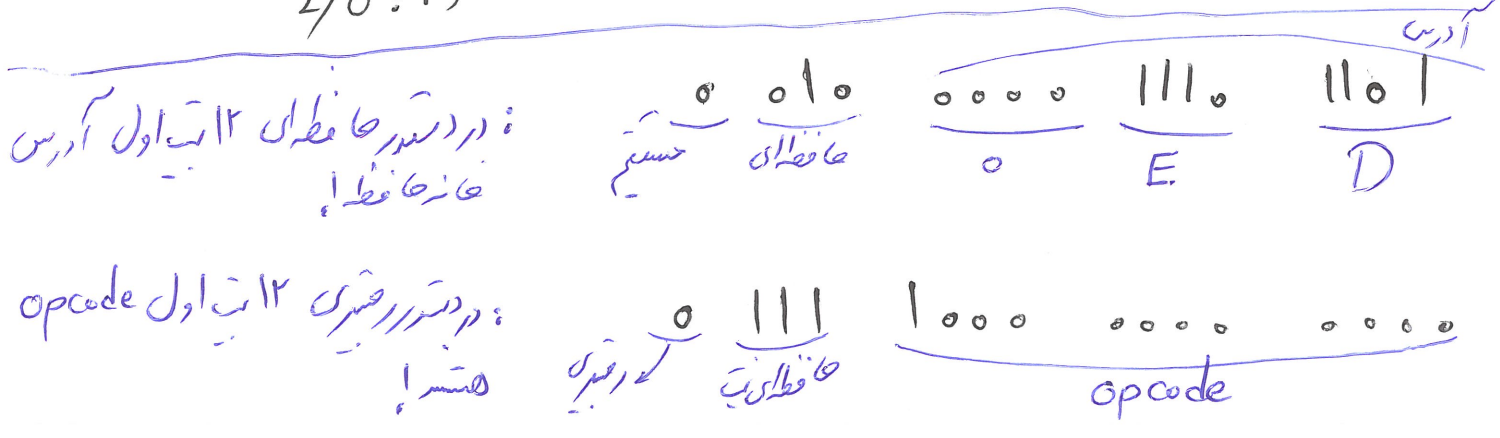




از راست:

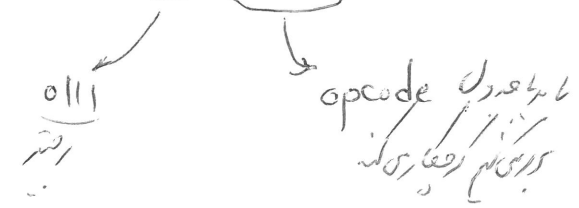




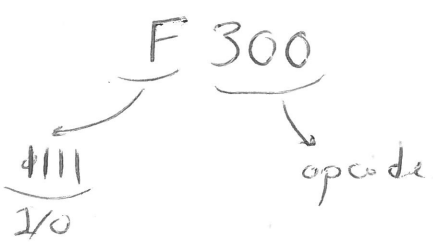


ماتریس جدول ۱۳۳ کتاب: دستورات حافظه‌ای: الف

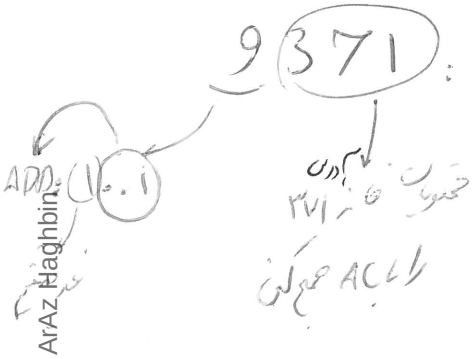
دستورات رجیستری:

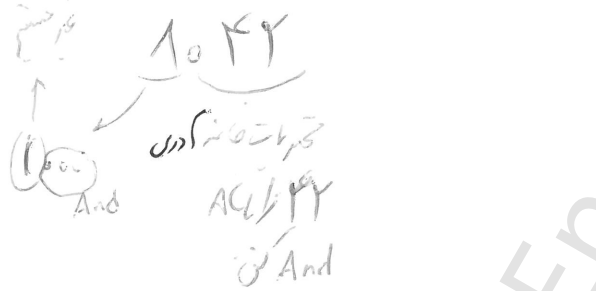


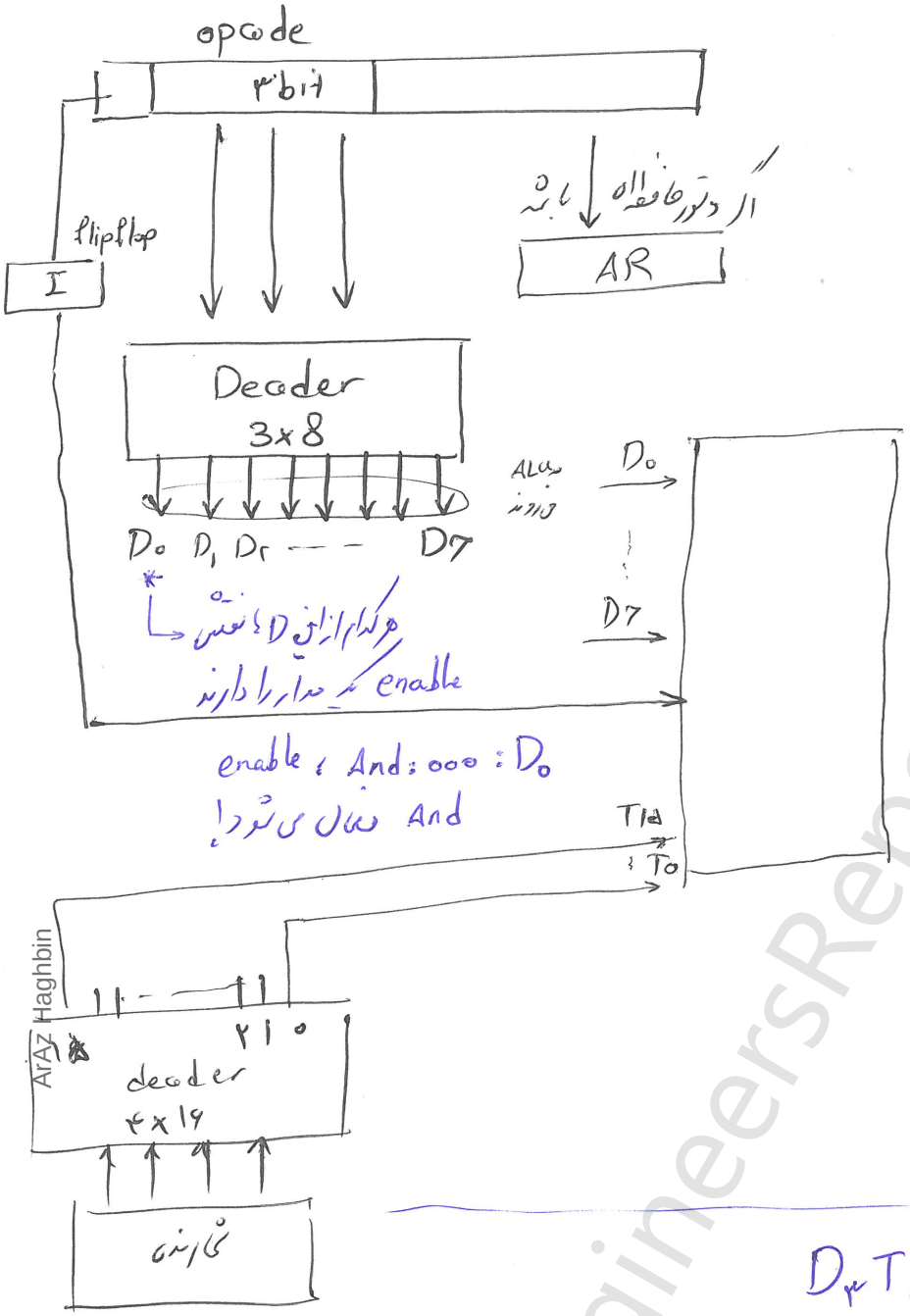
دستورات I/O:



ماتریس جدول صفحه ۱۳۴:







شکل صفحه ۱۳۸ :

D7 مشخص می‌کند که I چکاری کند!

شکل صفحه ۱۳۹ : $D_3T_4 : SC \leftarrow 0$

CLR
SC

مراحل اجرای دستورالعمل :

۱) Fetch

۲) Decode

۳) Address

۴) Execute

مثال: ارسال فرمان (عملیات):

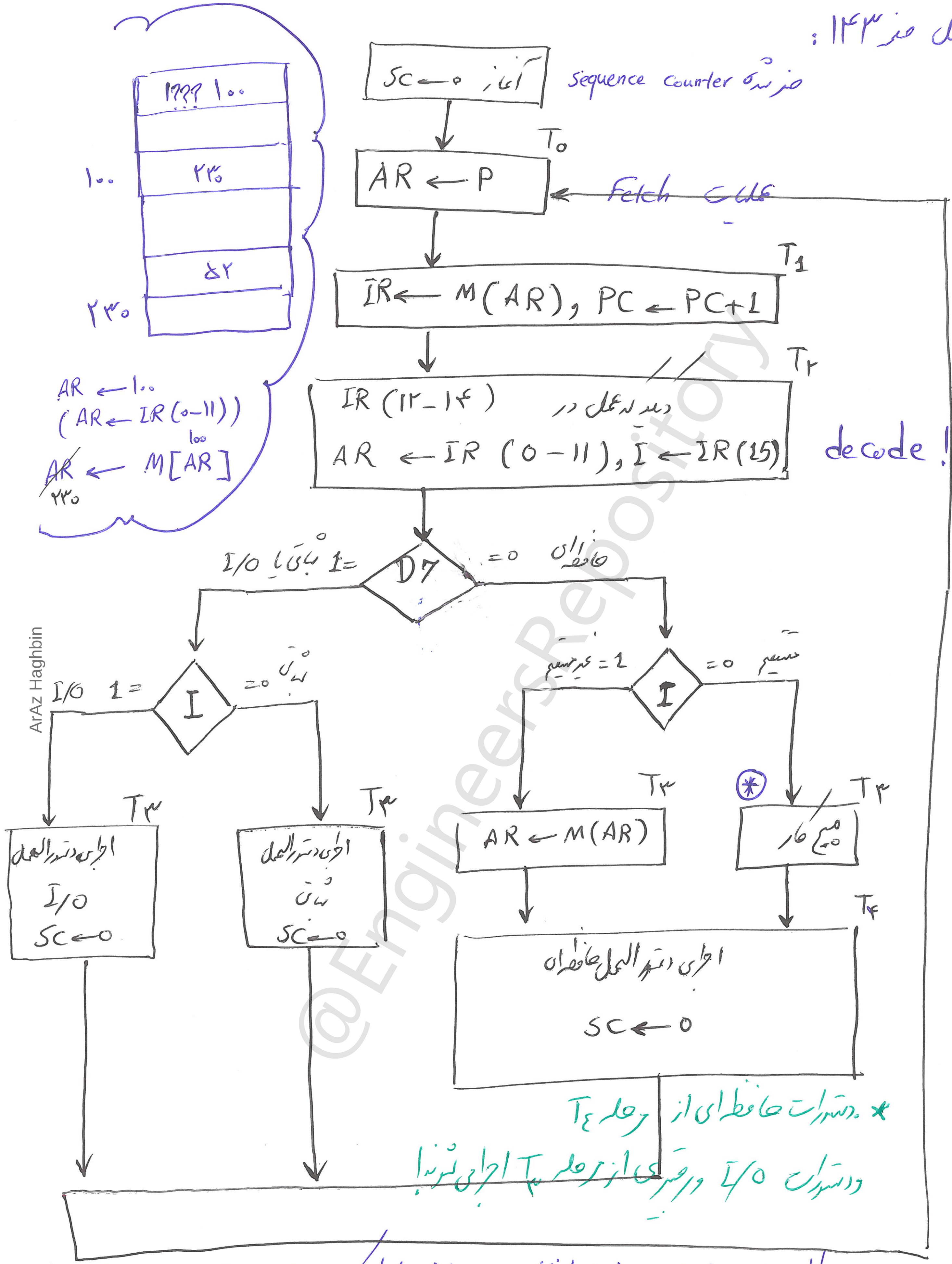
Fetch:
$T_0: AR \leftarrow PC$
$T_1: IR \leftarrow M[AR],\ PC \leftarrow PC+1$

decode:
$T_2: D_0 D_1 \dots D_7 \leftarrow IR(12-14)$ → decode از ۱۲ و ۱۳ و ۱۴ بیت
$,\ I \leftarrow IR(15)$ → بیت ۱۵ را بفرست به فلیپ فلاپ I
$,\ AR \leftarrow IR(0-11)$

۱- باید خواندن فعال شود
۲- مدیریت حافظه روی Bus : select
۳- باید IR load و ۴- باید PC INR
کدوم یکی از D_0 تا D_7 فعال شود

مثل صفحه ۱۴۲ توجه شود: این نرم‌افزار خاص مثل صفحه ۱۳۱ را نکشید!

شکل صفحه ۱۴۳:



* چون طراحی همه اجزای سخت‌افزاری و نرم‌افزاری باید قوانین را طبق آن دانسته باشیم (نمی‌دانیم مستقیم یا غیرمستقیم)

پس دستورات حافظه‌ای همگی از T_4 اجرا می‌گردند.

مراحل اجرای دستور: Fetch: برداشت از حافظه
decode: تحلیل
و

* در پایان هر دستورات Sequence counter برابر ۰ شده تا بتواند دوباره از اول دستور بعدی را اجرا کند!

جدول صفحه ۱۳۵: تمام دستورات رجیستری تمثل صفحه ۱۴۲:

نکته: برای اینکه در تمثل صفحه ۱۴۳ دستور رجیستری باشه باید ۳ شرط داشته باشه:

$D_7 = 1$
$I = 0$
در زمان T_3
$\longrightarrow D_7 \bar{I} T_3$

$\bar{I} = 1 \rightarrow I = 0$ ✓

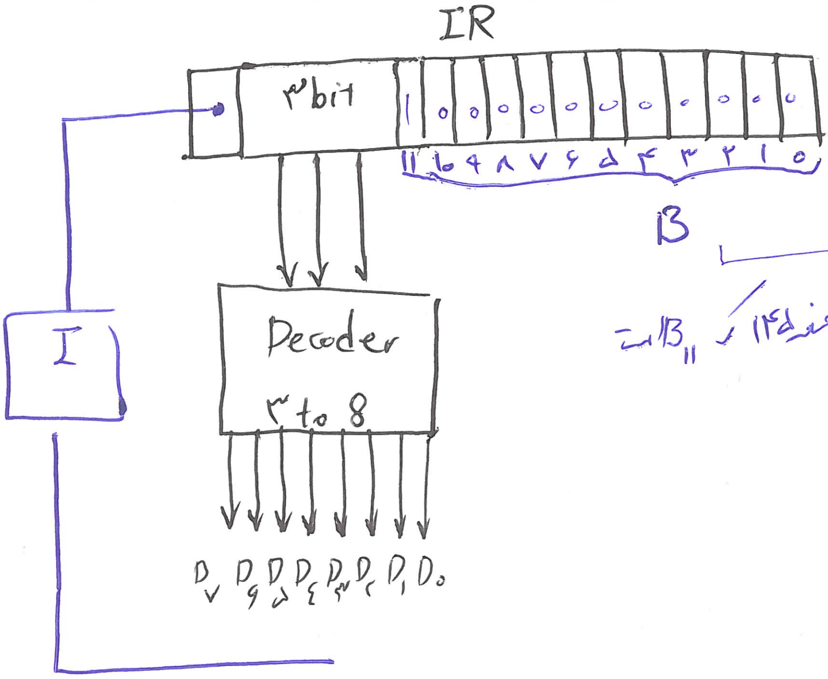
دستور I/O: $D_7 I T_3$

دستور حافظه‌ای: $D_7' T_4$

I نمی‌خواهد چون چه ۰ و چه ۱ دستور حافظه‌ای است و سوال مشخص نکرده (مشخص نکرده)

جدول صفحه ۱۳۵: صفحه بعد!

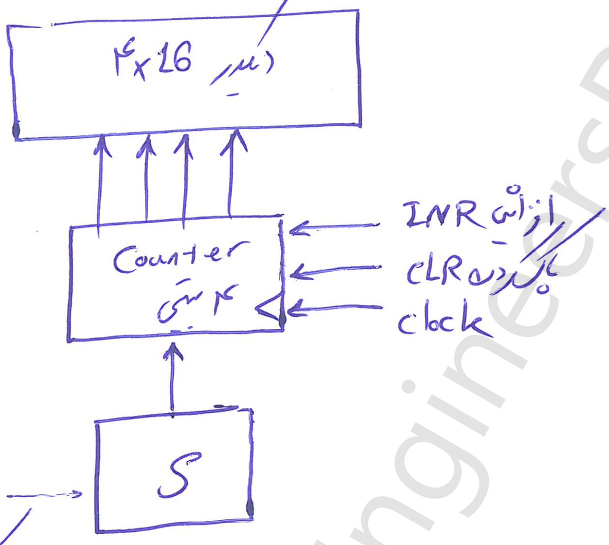
صفحه ۳ دارد!



مثلا برای CLA جدول صفحه ۱۴۵ که B_{11} است

* بیت آخر اگر ۰ باشد عدد مثبت و اگر 1 باشد عدد منفی است!

نکته:



باید صفر باید clock که با صفر اتفاق نیفتد که وقتی clock نباشد یعنی کامپیوتر خاموش است. (دکمه پاور اگر کامپیوتر در نهایت ورودی flip flop آن را 1 می کند!)

* جدول صفحه ۱۴۶ همان ۷ دستور حافظه‌ای طراحی شده در جلسه قبل را نشان می‌دهد!

مراحل fetch :

fetch { T_0 : AR ← PC
T_1 : IR ← M[AR] , PC ← PC+1

- مراحل decode :

decode { T_2: I ← IR(15) , $D_0 D_1 \cdots D_7$ ← IR(12-14) ,
AR ← IR(0-11)

IT_3 : AR ← M[AR] } پیدا کردن آدرس مؤثر

← شرط : flip flop I و (صفحه ۱۳۸) برابر 1 باشد!

T_3 و T_4 در همه دستورات مشترک است

جدول صفحه ۱۴۶ : (And) :

بعد از T_3 بالا ↓

And { D_0T_4 : DR ← M[AR]
D_0T_5 : AC ← AC ∧ DR , SC ← 0

چون مستقیم در AC نمی‌توان گذاشت

این D ها برای این است که کامپیوتر بفهمد کدام دستور را اجرا کند و از decoder آمده‌اند!

sequence counter = 0 می‌شود که عملیات را انجام دهد!

جدول صفحه ۱۴۶ : (ADD) :

ADD { D_1T_4 : DR ← M[AR]
D_1T_5 : AC ← AC + DR , E ← Cout , SC ← 0

فلیپ‌فلاپ برای carry خروجی ALU صفحه ۱۳۱

و صفحه ۱۴۶ : (load) : عملوندی را از حافظه برمی‌دارد و می‌گذارد در AC :

LDA : { D_2T_4 : DR ← M[AR]
D_2T_5 : AC ← DR , SC ← 0

ضمیمه دارد!

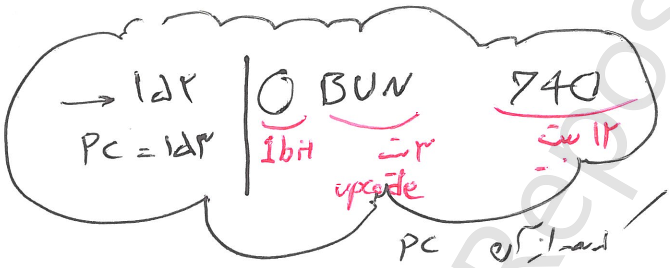
۱۳۹۲/۱/۱۱ (۷)

جدول صفحه ۱۴۶ : (STA) : مقدار ذخیره شده در AC را به AR انتقال داده M[AR] !

STA: { D_3T_4 : $M[AR] \leftarrow AC$, $SC \leftarrow 0$

جدول صفحه ۱۴۶ : (branch انشعاب) (BUN) : در حال اجرای دستور ۱۰۰ هستیم بعد از آن PC باید برود ۱۰۱ ولی خود ۱۰۰ دستوری دارد که برو مثلاً به ۳۷۵ (دستور ۳۷۵)

BUN: { D_4T_4 : $PC \leftarrow AR$, $SC \leftarrow 0$



PC بعد از این می‌شود 741!

جدول صفحه ۱۴۶ : (انشعاب با بازگشت) در برنامه‌نویسی هم توابع زیادی داریم که تابع وسط برنامه صدا می‌زنیم کار را انجام می‌دهد و دوباره ادامه کار (خود تابع)

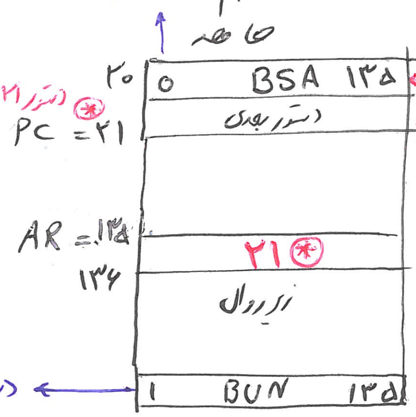
BSA

BSA: { D_5T_4 : $M[AR] \leftarrow PC$, $AR \leftarrow AR+1$

D_5T_5 : $PC \leftarrow AR$, $SC \leftarrow 0$

دستور ۱۰۰ می‌گوید برو به ۵۰۰ وقتی این اتفاق می‌افتد PC = 101 است برای اینکه وقتی برمی‌گردد برود به ۱۰۱ باید این ۱۰۱ جایی ذخیره گردد. (شکل صفحه ۱۴۹ الف)

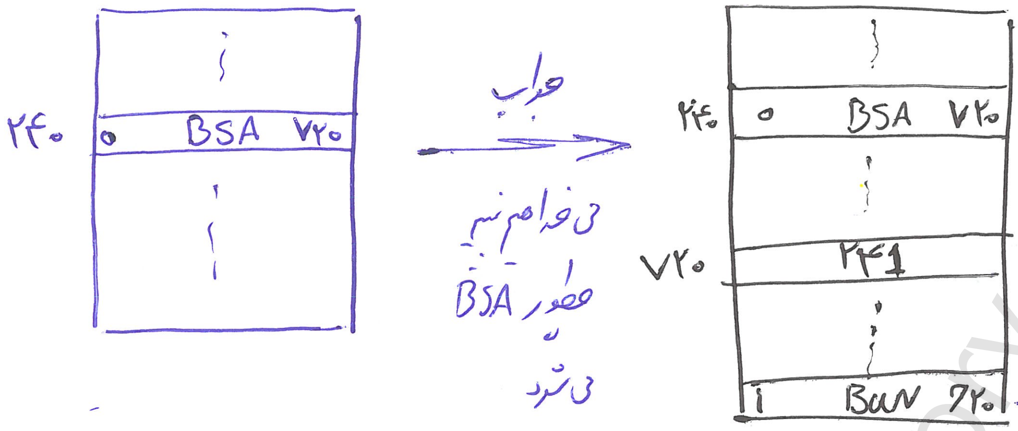
قصد رحله ریش



* دستور ۲۱ را در خانه ۱۳۵ می‌گذاریم تا دوباره برگردد

دستور برای رفتن به خانه ۱۳۵ و برداشتن دستور خانه ۲۱ برای بازگشت BSA (بدون بازگشت)

مثال : (امتحان)



* آدرس این را می‌دهیم چون می‌خواهیم جا عوض محتویات دیگر!

جدول صفحه ۱۴۶ : (ISZ) و خیلی کاربردی کرده index حلقه در برنامه‌نویسی یعنی i=0 تا i<10 کد این را کار می‌کند برای این کار . در حقیقت یک عدد منفی (-10) می‌گذاریم سپس با یک INC یک بار یک بار همان حلقه (یکی اضافه می‌کند) تا برابر 0 شود!

$$ISZ: \begin{cases} D_6T_4: & DR \leftarrow M[AR] \\ D_6T_5: & DR \leftarrow DR+1 \quad (Inc) \\ D_6T_6: & M[AR] \leftarrow DR,\ \text{iff } (DR==0),\ PC \leftarrow PC+1,\ SC \leftarrow 0 \end{cases}$$

عدد منفی

شرط index را چک می‌کند

* DR را انتخاب کردیم چون نمی‌توانیم مستقیم در AC بریزیم

* تنها عمل محاسباتی که بدون نیاز به ALU می‌توان باید رو انجام داد ، 1+ کردن (INC) می‌باشد!

* Intrupt. (صفحه بعد!)

* این D_4 برای این است که کامپیوتر تشخیص دهد کدام T_4 را باید اجرا کند که این D_4 را از decoder (صفحه ۱۳۸) می‌گیرد!

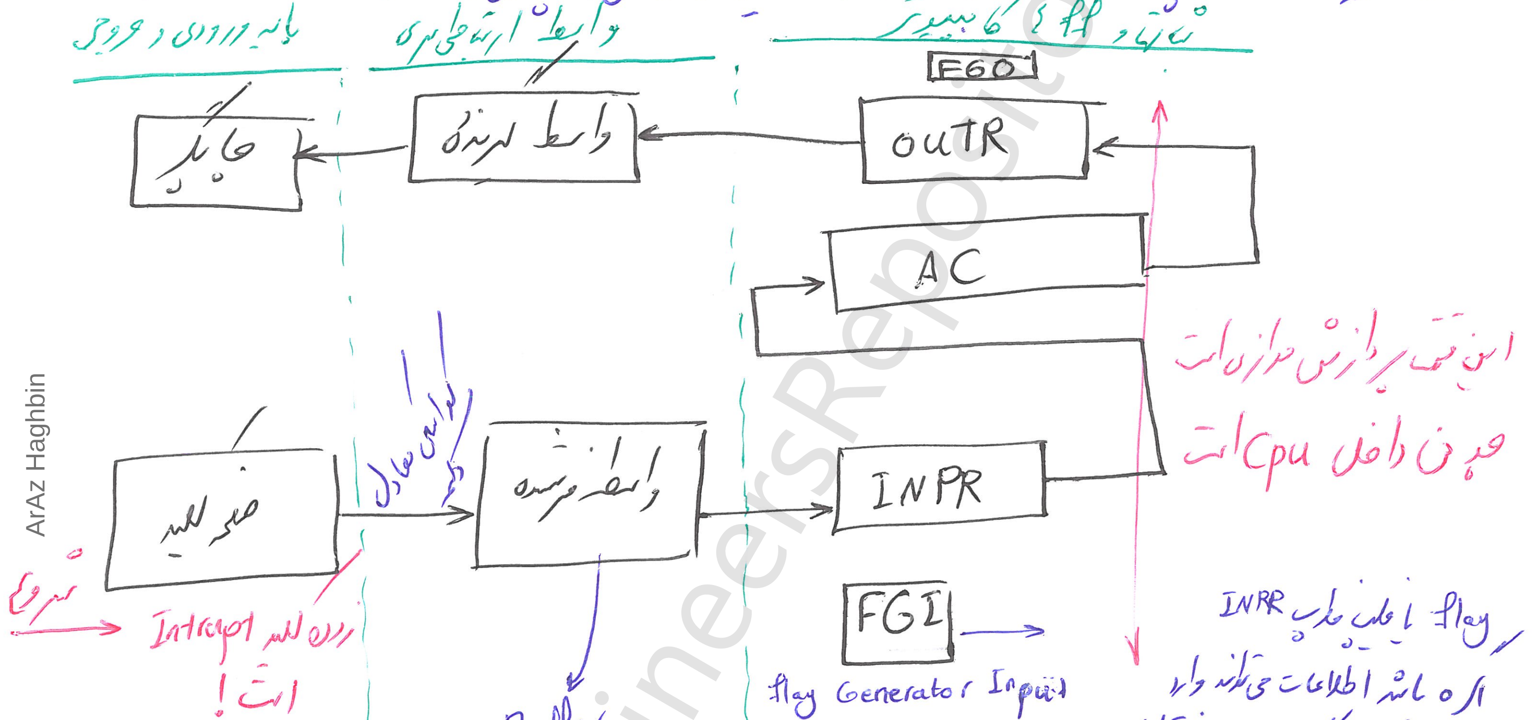
صفحه ۷ دارد!

*. cpu به دو روش به درخواستها پاسخ می دهد: Intrupt - pulling

pulling: مثل نگهبانی که مدام دارد چیزی را چک می کند (که با شخصات (ستاره کن) یعنی ISZ و pulling شبیه)

Intrupt: cpu کار عادی خود را انجام می دهد هر موقع لحظه خاصی آمد. (در حالتی که لازم به کار کاری کند) در زدن برای داده ها نیز به استفاده - با branch یا پرش)

شکل صفحه ۱۵۲: لحظه ای که وقتی کاربر دکمه ای روی کیبورد را می زند چه رخ می دهد!



*. فرض شده که از فرستنده به Buffer به صورت Serial انجام شده.

*. Buffer: حافظه کوچک است!

*. در هر طراحی اگر سرعت فرستنده و گیرنده با هم برابر باشد نیاز به Buffer است!

*. وقتی اطلاعات وارد INPR شد FGI ۱ می شود، اطلاعات دیگر وارد INPR نمی شود و در Buffer می ماند.

*. FGO اگر ۱ باشد می تواند OUTR اطلاعات را از AC بگیرد و بعد مثلا آن را به پرینتر می دهد و بعد از آن دوباره ۱ می شود تا تاثیر داد AC!

(بین FGO صفر می شود)

نکته مهم در صفحه بعد!

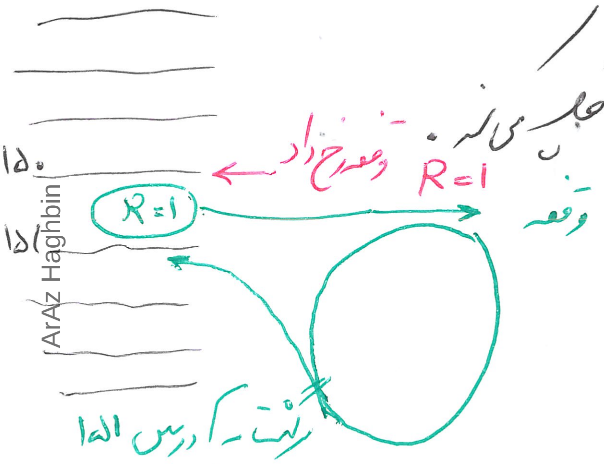
در شکل عملکرد کیبورد تمام ارتباطات بین AC با دو رجیستر INPR و OUTR

به صورت سری است و فقط بین AC با دو رجیستر INPR و OUTR عملیات جابجایی داخل CPU اتفاق می افتد.

cpu :
- pulling : cpu خودش مدام دنبال یا منبع مورد نیازش می‌گردد .
- Intrupt : خود برنامه - یا دستور قطع می‌کند .

صفحه ۱۸۸ : شکل بسیار مهم!

* . کامپیوتر اگر در حال اجرای دستور (دستور فعلی) باشد و وقفه جواب نمی‌دهد ، اول دستور را اجرا می‌کند و سپس وقفه را جواب می‌دهد .



** . کامپیوتر در پایان هر دستور flipflop - R را چک می‌کند .

*** . اگر به دستور بعدی رجوع کردیم یعنی باید جایی آن را ذخیره کرده باشیم! (مثل BUN - BSA)

*** . IEN (Intrupt enable) از :

R → ۰ وقفه رخ نداده
R → ۱ وقفه رخ داده

IEN → ۰ وقفه‌ها اجازه ندارد وارد شود
IEN → ۱ وقفه‌ها اجازه دارد وارد شود

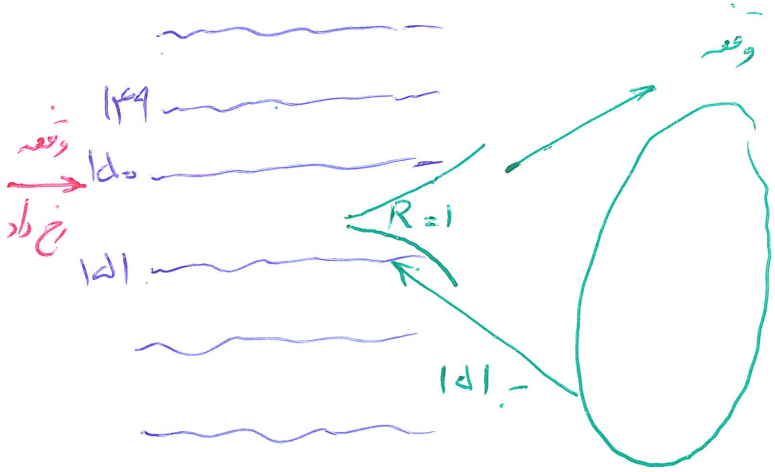
**** . اولویت IEN از R بسیار بالاتر است!!!

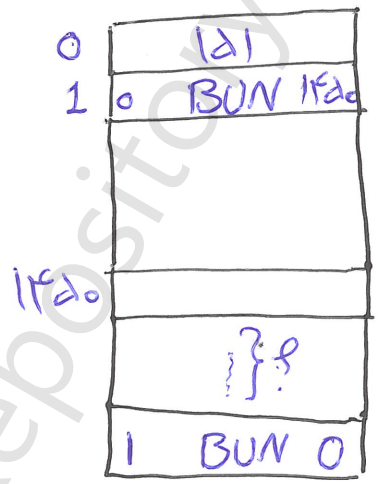
* . اگر FGI و FGO هر دو (همزمان) یک باشند اولویت با FGI است .

* . روال وقفه بسیار شبیه BSA است ولی BSA می‌گوید هر جایی (شخص ذخیره کن) اما وقفه می‌گوید در خانه ۰ (صفر) ذخیره کن!

* - در وقفه محل ذخیره آدرس بازگشت فقط و فقط در خانه ۰ (صفر) است ول در BSA خودمان به آن می‌گوییم کجا برود!

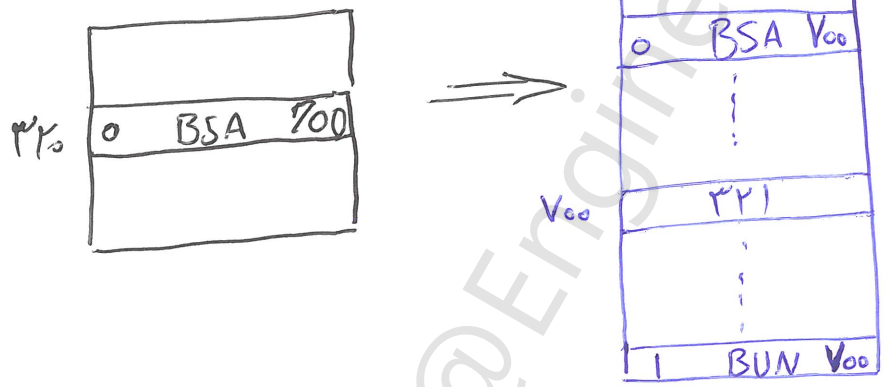
* - خود وقفه مثل یک برنامه مجموعه‌ای از دستورات است.



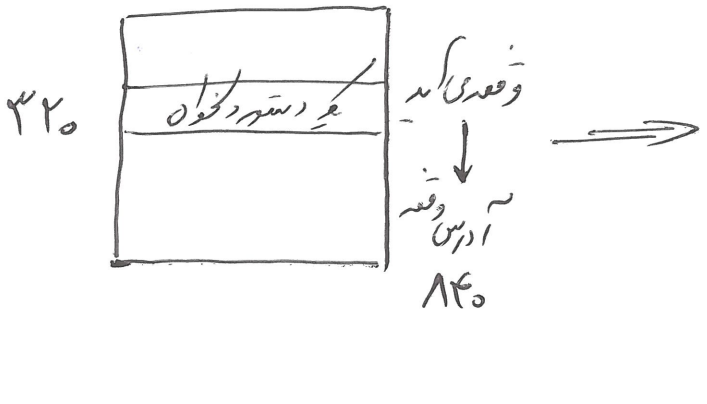


تفاوت وقفه‌ها BSA:

BSA:



وقفه:



صفحه ۳ دارد!

* در آموزش دستور وقفه IEN = 1 می‌شود!

وقفه‌هایی که تا اینجا دیدیم:

$$R \leftarrow 1$$

$IEN = 1$

$FGI = 1$ یا $FGO = 1$

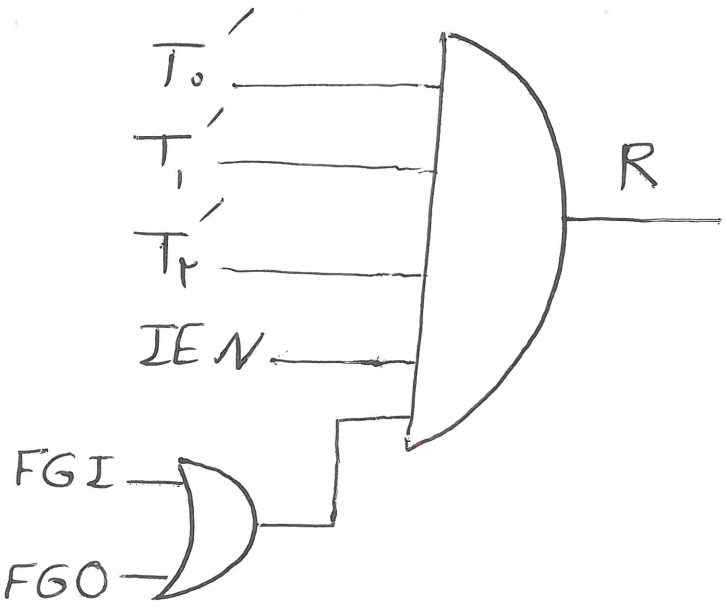
T_0' و T_1' و T_2' در زمان T_0, T_1, T_2 ساعت

$T_0 = 0 \Rightarrow T_0' = 1$

$$R \leftarrow 1: \quad T_0' T_1' T_2' \cdot IEN \cdot (FGI + FGO) \quad (*)$$

And

یعنی حاصل این جمله 1 باشد!



تا اینجا دیدیم

۶ وقفه

فصل ۸ : صفحه ۲۴۲ : پردازنده با V و AC (Accumolator) با R_1 تا R_7

مثال : 01000 ← ۵ بیت چون ۵ opR داریم!

$R_5 \leftarrow R_2 + R_3$

- R_5 → Sel D → 101
- R_2 → Sel A → 101
- R_3 → Sel B → 011

قالب دستورالعمل :

Sel A	Sel B	Sel D	OPR
3	3	3	5

۱۴ بیت

جدول صفحه : ۲۴۵ با توجه به شکل ۲۴۲ و جدول ۲۴۴ :

$R1 \leftarrow R2 - R3$ ⟶ 010 011 001 00101

$R4 \leftarrow R4 \vee R5$ ⟶ 100 101 100 01010

⋮
ادامه در کتاب

روش سوم (روش اول : تک AC و روش دوم چند AC)

استفاده از پشته (Stack) : صفحه ۲۴۷ : به جای چند AC یک Stack که داخل Cpu است.

* هر کمیتی در Stack می خواهیم بگذاریم (push) باید اول در DR بگذاریم و اگر بخواهیم از Stack برداریم (Pop) باید باز از DR استفاده کنیم.

صفر که دارد!

برای push کردن در Stack :

Push:

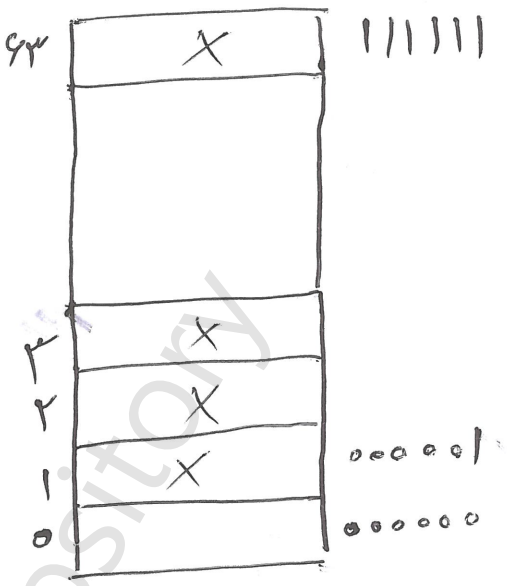
$SP \leftarrow SP+1$

$M[SP] \leftarrow DR$

if $(SP == 0) \rightarrow Full = 1$

$empty = 0$

چون ۶۴ تا ۷ رقمی نشد ولی چون تا ۶۳ می‌شد ۶ رقمی است یعنی ۶ رقم می‌شد



برای Pop کردن :

$DR \leftarrow M[SP]$

$SP \leftarrow SP-1$

if $(SP == 0) \rightarrow Empty = 1$

$Full = 0$

دستورات چند آدرسی :

سه آدرسه : $R1 \leftarrow R2 + R3$

③ ② ①

دو آدرسه : $R1 \leftarrow R1 + R2$

① ②

ارجاع به بیلیگ (یعنی Ac حامل عملوند و جواب است) تک آدرسه : $Ac \leftarrow Ac + DR$ ①

استفاده از آن ندارم ولی DR متغیر

stack { pop x → از بالای پشته بردار ⊛ ; push X } : صفر آدرسه (آدرس مشخص نداده ام)

⊛ از بالای پشته بردار بگذار در X

$X = (A+B) * (C+D)$

مثال در Cpu فصل ۵ (با Ac) :

روش اول با Cpu Ac :

$Ac \leftarrow M[A]$

$Ac \leftarrow AC + M[B]$

$M[T] \leftarrow A$ (temp)

$Ac \leftarrow M[C]$

$Ac \leftarrow Ac + M[D]$

$Ac \leftarrow Ac * M[T]$

$M[X] \leftarrow AC$

نکته : کامپیوتری رش از چند آدرسی (۲، ۳، ...) داشته باشد همانش از AC کردار!

حل: همان مثال قبل به صورت ۲ آدرسه (یعنی ۲ AC داریم):

$R1 \leftarrow M[A]$

$R1 \leftarrow R_1 + M[B]$

$R2 \leftarrow M[C]$

$R2 \leftarrow R2 + M[D]$

$R1 \leftarrow R1 * R2$

$M[X] \leftarrow R_1$

تو هر دوشون از یه AC داریم
دیگه نیازی به temp نیست!

۶ دستور

حل: همان مثال به صورت ۳ آدرسه:

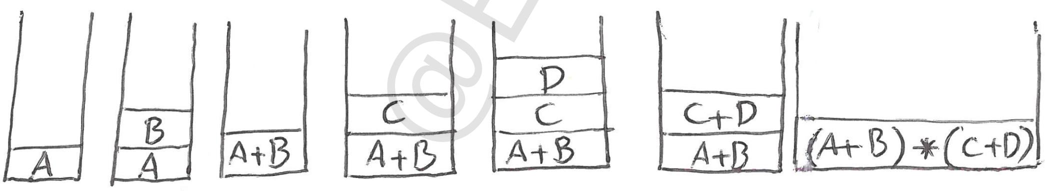
$R1 \leftarrow M[A] + M[B]$

$R2 \leftarrow M[C] + M[D]$

$M[X] \leftarrow R1 * R2$

۳ دستور

حل: همان مثال به صورت ۰ آدرسه: (۰ آدرسه از پشته استفاده می‌کند)



pop اول هر دو را
می‌کنه و بعد از جمع کردن
push می‌کنه!

push A
push B
ADD
push C
push D
ADD
Mul
POP

صفحه ۳ را رد!

دو روش معماری برای طراحی cpu داریم:

- RISC : Reduced instructions Set computer
- CISC : Complex instructions Set computer (دستورات پیچیده)

ایرادات CISC : ۱- به علت طراحی پیچیده گران قیمت‌اند

۲- به علت مصرف انرژی بالا چون مدارات زیاد دارند مناسب دستگاه‌های موبایل و سیستم باتری‌دار نیست

۳- به علت مصرف انرژی بالا حرارت زیادی ایجاد می‌کند که به سیستم خنک‌سازی بیشتر نیاز دارد

مزایای CISC : اینکه فقط با یک دستورالعمل عملیات مورد نظر (مثلاً مثال ضرب) را انجام می‌دهد

در دنیای واقعی Intel و AMD اکثراً CISC هستند.

ایرادات RISC : ۱- دستورالعمل‌ها را در تعداد بیشتری انجام می‌دهند

مزایا: مصرف انرژی کم که آن را مناسب دستگاه‌های موبایل می‌سازد - حرارت کمتری ایجاد می‌کند و ارزان‌ترند

اکثر دستگاه‌های موبایل موجود از معماری RISC پیروی می‌کنند.

* بزرگترین مشکل کامپیوتر در دسترسی cpu به حافظه است! حافظه Bottle_neck است یعنی سرعت را پایین می‌آورد

** RISC تمام اطلاعات مورد نیاز خودش را اول از حافظه می‌گیرد و در AC یعنی (accumulator) خودش می‌گذارد سپس کار خودش را شروع می‌کند. معماری cpu های RISC شامل تعداد بالایی AC است. (یعنی اگر قرار است دستور ۱۰۰ را اجرا کند تا دستور ۱۴۰ را قبلاً برداشته مثلاً عملگر ۲ pc را اشغال کند) عملگر (operator)

* کاهش تعداد دستورات به معنای افزایش سرعت نیست!

RISC:

خواندن عملوندها:

$R1 \leftarrow M[A]$

$R2 \leftarrow M[B]$

$R3 \leftarrow M[C]$

$R4 \leftarrow M[D]$

این قسمت کاملاً موازی و همزمان

wintel: windows → Intel

محاسبات:

$R1 \leftarrow R1 + R2$

$R3 \leftarrow R3 + R4$

$R1 \leftarrow R1 * R3$

در نهایت در یک سیکل یا ۲ سیکل

$M[x] \leftarrow R1$

نوع آدرس دهی | op code | Address |

روشهای آدرس دهی: - آدرس دهی مستقیم

- آدرس دهی غیر مستقیم

(دقت شود مثل pop نیست) - آدرس دهی ضمنی (مثل دستور pop (Stack)) یعنی خود دستور مشخص می‌کند آدرس کجاست

- آدرس دهی بلافصل (بلافصل): $Ac \leftarrow Ac + 10$؛

در این آدرس دهی، این آدرس نیست

مثلاً 01 ADD 10

این مقدار است نه آدرس

- آدرس دهی رجیستر (مستقیم) جزء data (عملوند)

- آدرس دهی رجیستری (غیر مستقیم) آدرس — حکم کار دارد

- آدرس دهی خودافزایشی: از مقدار فعلی استفاده می‌کند بعد یکی به آن اضافه می‌کند R_1 [400] → بعد R_1 [401]

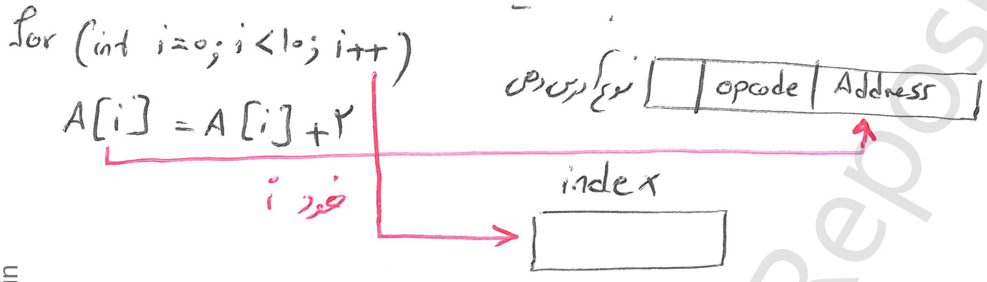
- آدرس دهی خودکاهش: اول از مقدار یکی کم می‌کند بعد استفاده می‌کند

R[۴۰۰] —اول→ R[۳۹۹] —بعد استفاده→

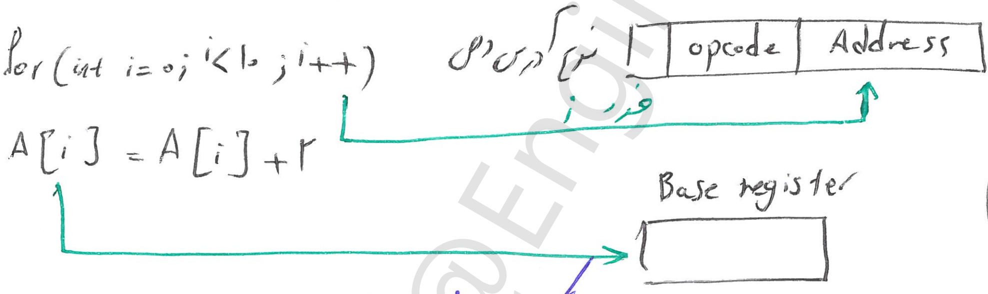
- آدرس دهی نسبی: Field آدرس با PC جمع می‌شود آدرس جدیدی بدست می‌آید که روی آن data قرار دارد.

- آدرس دهی شاخص دار: Field آدرس با رجیستر index جمع می‌شود آدرس جدیدی بدست می‌آید که روی آن data قرار دارد.

مثال: در حلقه‌های برنامه نویسی:



- آدرس دهی با ثابت پایه: آدرس با Base register جمع می‌شود آدرس جدیدی بدست می‌آید و داده در آن آدرس.



مثال صفحه ۲۶۴

** نکته: cpu های اینتل از آدرس دهی شاخص دار استفاده می‌کنند معمولاً از آدرس دهی با ثابت پایه استفاده نمی‌کنند و بالعکس

* آدرس دهی موثر: آدرسی که دیتا از آن برداشته می‌شود.

جدول صفحه ۲۶۵ !

دستورات کنترل: صفحه ۲۷۶ کتاب:

- BUN
- BSA

} قبلاً مطرح شده

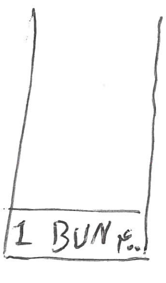
- BR (Branch) انشعاب { شرطی: شرطی بررسی می‌شود اگر درست بود انشعاب رخ دهد
  غیرشرطی: بدون هیچ شرطی انشعاب رخ دهد (BUN غیرشرطی است) اگر آمده ی رود به انشعاب

- JMP (Jump) انشعاب: در cpu های مختلف برخی branch را انشعاب برای
  برخی cpu ها jump می‌گویند.

* در برخی cpu ها هم jump و هم branch داریم. به شرطی jump و به غیرشرطی branch گویند. اینها فقط تعریف اسم هستند.

- Call: همان BSA است

- Return (RET): اون دستوری که پایین interrupt یا BSA انجام می‌شود (همان BUN پایان BSA)



Ret (RET)

- CMP: (Compare) مقایسه دو مقدار که هم برابرند یا نه از هم تفریق می‌کند اگر حاصل ۰ شد یعنی برابرند

- TST: اگر این دو شرط برقرار بود این کار را انجام بده فرضی

صفحه ۷ دارد!

ادامه دستورات کنترل :

- SKP : (skip)

مثال :

```
if ( x>10)
    f1();
else
    f2();
```

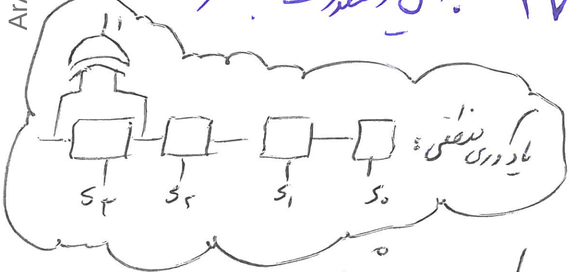
در skip :

```
SKP  x>20
BUN  f2( );
f1( );
```

یعنی اگر شرط درست باشه از روی دستور بعدی بپره یعنی از روی PC بعدی می‌پرد

* اگر شرط درست باشه می‌پرد اگر غلط باشه نمی‌پرد

رجیستر وضعیت (Status Register) شکل صفحه ۲۷۴ : این دستورات بالا را مشخص می‌کند (رجیسترها)



* overflow : یعنی عملیات حسابی به خطا رخ می‌دهد !

* Alu هر کاری که انجام بدهد که رجیستر وضعیتی update می‌شود

جدول صفحه ۲۷۵ !

صفحه ۲۷۷ کتاب : آدرس بازگشت

یک روش : ذخیره آدرس بازگشت در یک آدرس از قبل تعیین شده ! همان وقفه

مثال : ۱۴۱ :

PC : ۱۴۲ (محل ذخیره آدرس بازگشت) ← در وقفه ۱۴۲ را در خانه ۰ می‌گذارد

روش دوم : ذخیره آدرس بازگشت در آدرسی که توسط دستور تعیین شده ! همان BSA

→ ۱۴۱ : 0 BSA 760

PC : ۱۴۲ :

روش سوم : ذخیره آدرس بازگشت در یک Register از قبل تعیین شده ! مثل : مثلاً R5

روش چهارم : ذخیره آدرس بازگشت در پشته .

نکته : پشته از همه بهتر است (روش سوم) چون که تنها با پشته می‌توان چند انشعاب را همزمان پاسخ داد .

* . وقتی وقفه رخ می‌دهد قبل از شروع به اجرای دستورات وقفه چه کارهایی باید انجام دهیم؟

۱ـ ذخیره PC در آدرس ۰ (ذخیره آدرس بازگشت در خانه تعیین شده)

Ac = 5
DR = 7
TR = ۱۳

۲ـ ذخیره محتویات تمام رجیسترهایی که پردازنده (که بعد از انجام وقفه برای برگشتن درون خودشان ؛ مثلاً کامپیوتر فکر می‌کند که اصلاً به وقفه نرفته .)

۳ـ ذخیره رجیستر کنترل ص ۲۷۴ (PSW) : مثلاً C S V Z

* . cpu وقتی وقفه می‌آید یک scan از محیط خود انجام می‌دهد و آنها را در خانه‌های حافظه (RAM) ذخیره می‌کند.

صفحه ۲۸۱ : انواع وقفه :

* . وقفه خارج و داخل با توجه به cpu سنجیده می‌شود مثلاً خارجی یعنی یک (خارج بیرون از cpu)

مثال‌های وقفه خارجی : عملیات I/O : زدن دکمه keyboard ، موس ...

اعلام پایان عملیات I/O : ارسال پیام پایان دریافت پرینت از پرینتر به cpu

Time-Out : مثل not responsive شدن برنامه در ویندوز ، کامپیوتر برای هر برنامه‌ای یک time در نظر می‌گیرد (اینکه timeout می‌گذارد) که این زمان را cpu در نظر می‌گیرد.

مدیریت منابع انرژی : مثل باقی ماندن اطلاعات بعد از آلارم باتری و خاموش شدن.

وقفه داخلی : از درون cpu رخ می‌دهد : ۱- استفاده اشتباه از دستور (یا غیرمجاز - عمدی) مثل: overflow شدن - خطای تقسیم بر صفر - شیفت‌های بیجا

وقفه نرم‌افزاری : وقفه‌ای که توسط برنامه‌نویس انجام می‌گیرد : ۳ عمل بعد از رخ دادن وقفه که خود برنامه‌نویس صورت می‌دهد (در سطح پایین یعنی سخت‌افزار صورت می‌گیرد)

صفحه ۳ دارد!

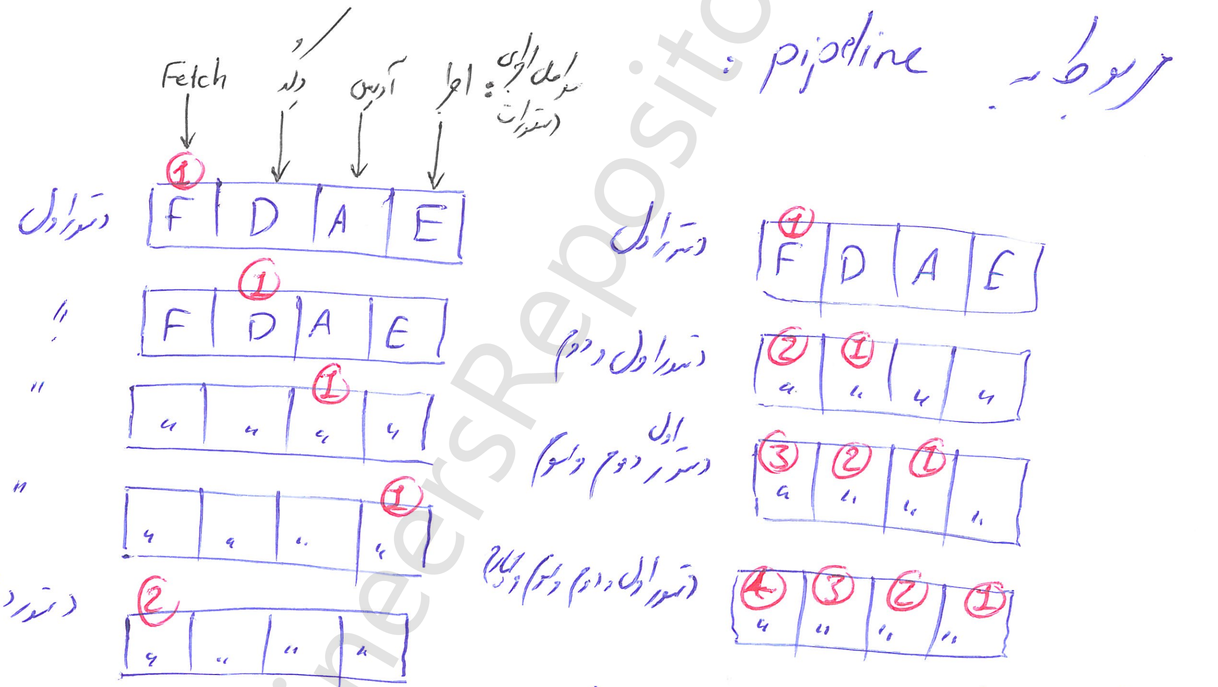
* . cpu های RISK در حین اجرای عملیات سراغ حافظه نمی روند!

صفحه ۲۸۳ الی ۲۸۴ : مقایسه RISK و CISC :

* . دستورات با طول مختلف در decode کردن مشکل ایجاد می کند. (مشخصات CISC)
راندمان pipline بیشتر در decode پایین می آید.

پایپ لاین = pipeline
ماشینی را می توان
موازی: چند کار را همزمان و
نا ترتیبی: نا ترتیبی انجام داد.

رسم جدول pipeline:

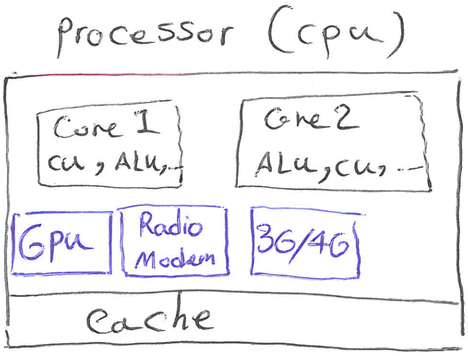


* . واحد کنترل cpu های RISK (RISC) کاملا سخت افزاری است:

۱ـ مزیت: سرعت بالا

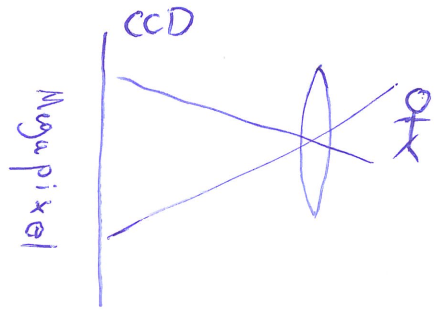
۲ـ ایراد: چون کاملا سخت افزاری است امکان اعمال تغییرات در آن وجود ندارد.

تفاوت بین cpu و هسته چیست؟ مثلا در cpu دو هسته ای:
تعداد هسته بیشتر به معنای cpu بهتر نیست ـ معماری داخل هسته نشان از عملکرد کلی (راندمان کاری) است.



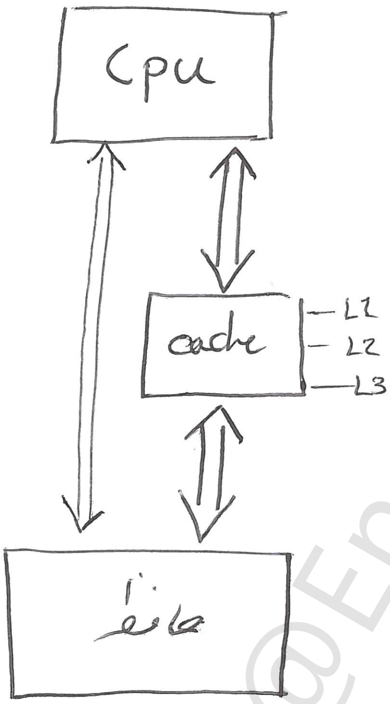
→ SOC : System on chip

کیفیت دوربین : CCD : حفره‌ای که عکس گرفته شده روی آن ذخیره می‌شود هرچه بزرگتر بهتر



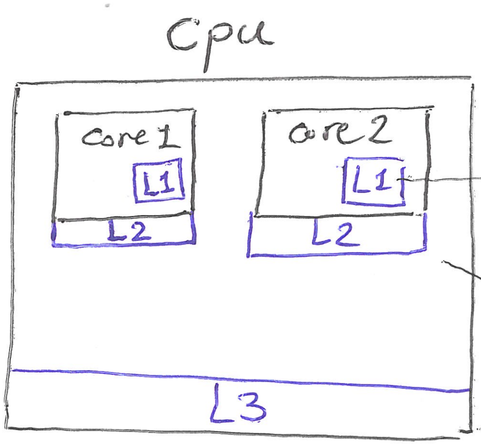
CCD 22 Mpixel
22 میلیون نقطه
هرچه نقاط بزرگتر شوند
کیفیت بهتری می‌یابند.

* ارتباط بین cpu و حافظه بار زیاد است. یکی از راه‌های حل این مشکل استفاده از حافظه cache است که سرعتی بسیار بالا دارد. cpu اول اطلاعات مورد نیاز خود را در cache جستجو کند اگر پیدا نکرد به حافظه می‌رود. اگر اطلاعات مورد نیاز را در cache پیدا کند می‌گویند hit شده و اگر پیدا نکرد miss شده.



hit rate : به میزان درصد پیدا کردن اطلاعات مورد نیاز cpu از cache گویند که در معماری‌های امروزی بالای ۹۵٪ است.

انواع cache : (از نظر محل قرارگیری)

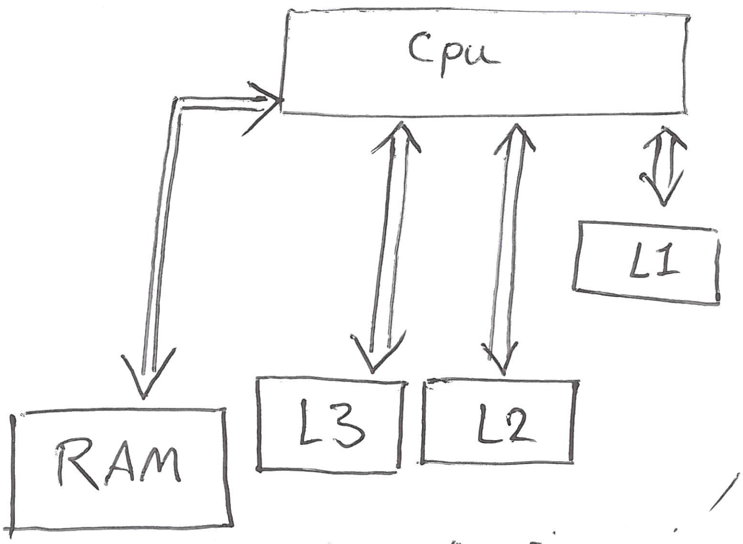


L1 (۳۲kB – ۶۴kB): حافظه‌ای کوچک با سرعت بسیار بالا قرار گرفته در هسته با سرعت دسترسی بسیار زیاد.

L2 (۱۲۸kB – ۵۱۲kB): چسبیده به هر core ولی نه داخل آن

L3 (۲MB – ۱۶MB): داخل cpu که به صورت اشتراکی توسط همه core ها استفاده می‌شود

ترتیب دسترسی cpu به cache ها:



* در RAM هر بیت از یک تا دو ترانزیستور و یک خازن لازم یا فرض می‌شود
ولی در cache هر بیت از یک حداقل ۶ ترانزیستور (بدون خازن) نیاز دارد که ساخت آن
در ابعاد RAM (16GB و 8GB و ...) بسیار حجیم و گران خواهد شد.

---

* درباره‌ی توضیح انواع cache !

مثال: اگر سرعت دسترسی به حافظه 1000 ns (نانو) باشد و سرعت دسترسی به cache 100ns باشد و از هر 10 درخواست 9 تای آنها در cach یافت شود متوسط دسترسی به اطلاعات چقدر است؟

$$\frac{\underbrace{9 \times 100}_{cache} + 1 \times (\underbrace{100}_{\text{اول cache را گرد}} + \underbrace{1000}_{\text{حافظه}})}{\underbrace{10}_{\text{متوسط}}} = \frac{2000}{10} = 200\,ns$$

فرض با L1، L2، L3:

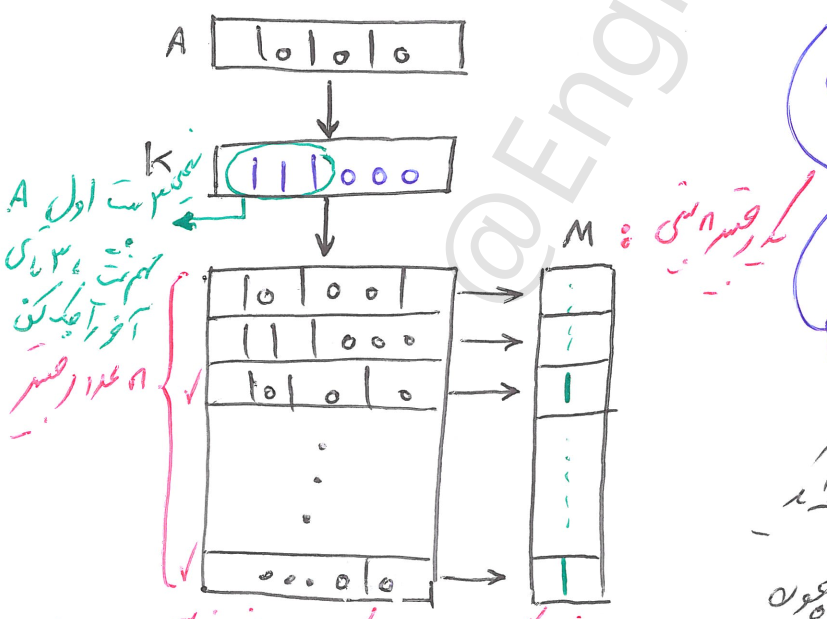
سرعت متوسط؟

		از هر ۱۰ درخواست
سرعت دسترسی به حافظه	1000 ns	۱
L3 " "	150 ns	۲
L2 " "	120 ns	۲
L1 " "	10 ns	۵

$$\frac{\underbrace{5\times10}_{L1}+\underbrace{2(10+120)}_{L2}+\underbrace{2(10+120+150)}_{L3}+\underbrace{1(10+120+150+1000)}_{\text{حافظه}}}{10}$$

حافظه‌ی Associative (تداعی) صفحه ۴۵۷ کتاب:

سرعت بالا و بار گران در پیاده سازی که بر اساس آدرس نیست و از حافظه معمولی بیشتر هزینه دارند!



← اگر در رجیستر پیدا کرد بیت متناظر آن در M را 1 می کند.

صفر ۷ دارد!

مثل پرسیدن اسم از هر کلاس تا پیدا شود (حافظه معمول)

یعنی گشتن خانه به خانه

صدا زدن اسم که بلند شود (حافظه تداعی گر)

* این ساختار حافظه به ازای هر خانه حافظه یک مقایسه گر دارد، که همه همزمان عمل می کند چون مقایسه گرها مستقل هستند.

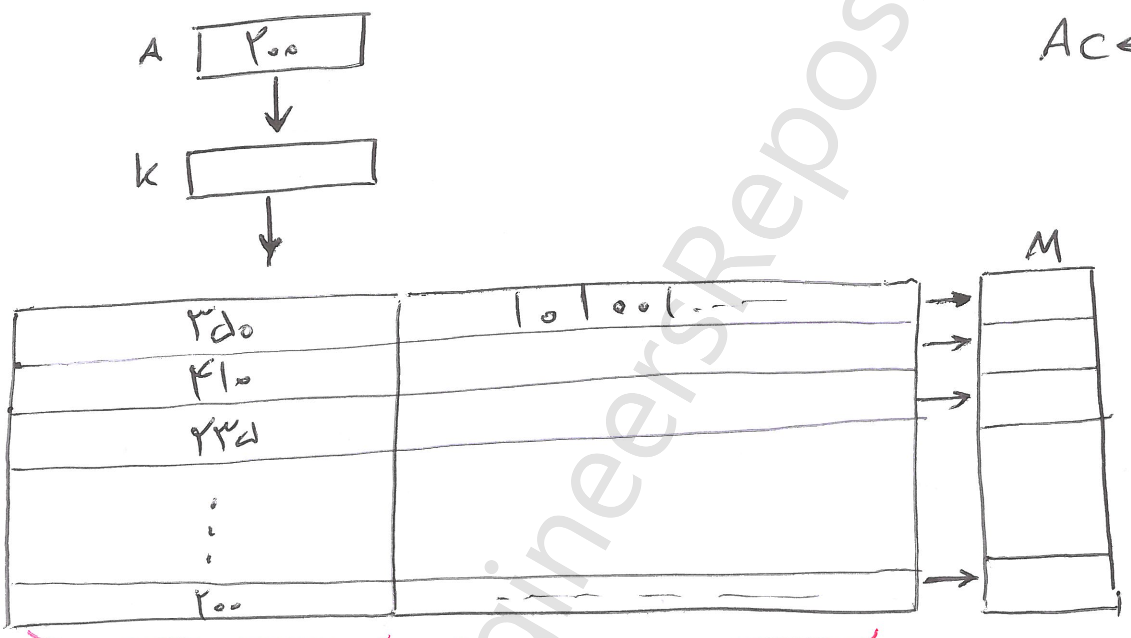
معماری cache (روش ساخت آن که ۳ روش است) از صفحه ۴۶۵ تا صفحه ۴۷۱ کتاب

روش ۱: cache های تداعی‌گر: (شکل صفحه ۴۶۶)

سوال: می‌خواهیم بدانیم این شکل صفحه ۴۶۵ کش در نهایت چند بیتی می‌شود!

* برای شکل صفحه ۴۶۵ داریم:

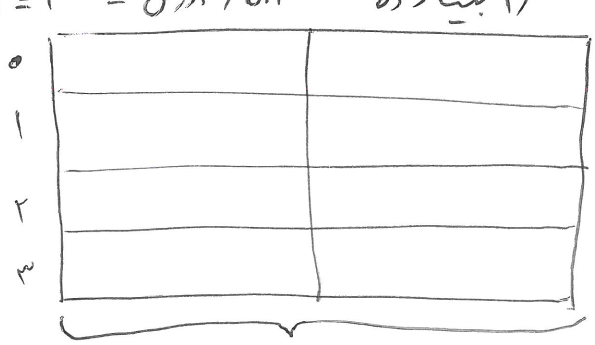
$Ac \leftarrow Ac + M[200]$



این عمل را mapping گویند!

۳۲k = ۲^۱۵ = 15 bit + 12 bit = ۲۷ bit نیاز دارد!

چون در شکل صفحه ۴۶۵ ۳۲k خانه داریم



مثال: حافظه: ۶۴ خانه ۱۶ بیتی

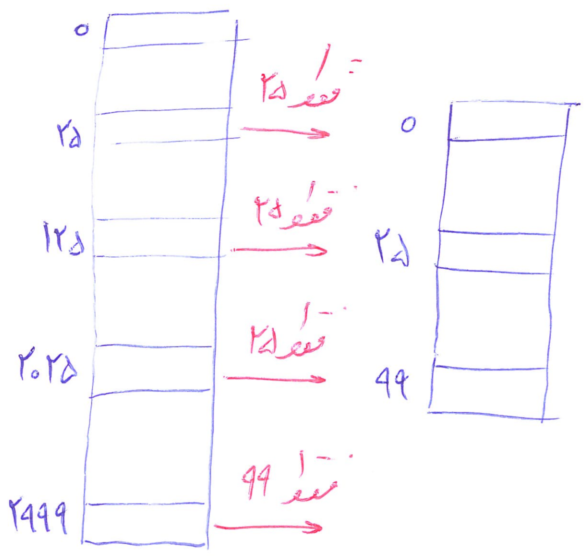
کش: ۴ خانه دارد ۴ بیتی

جواب: کش ما ۴ خانه ۲۲ بیتی دارد!

۲۲ = ۶ + ۱۶ بیت نیاز است!

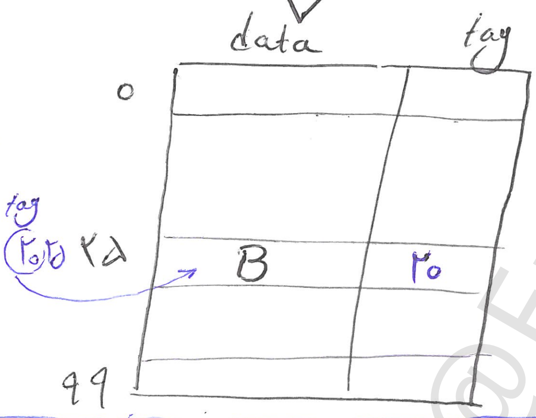
روش دوم: نگاشت مستقیم: ص ۴۶۶ انتهای صفحه: در این روش تنها (یک cache) آدرس دارند.

مثال Decimal:



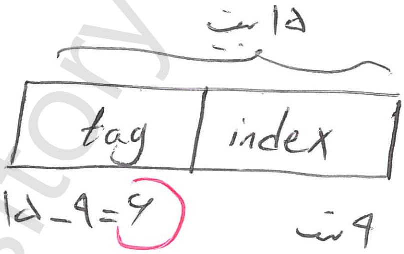
* دایرکت mapping از همین فرمول استفاده می‌کنند.

مثلا



با توجه به مثال کتاب:

$۳۲k = ۲^{۱۵}$

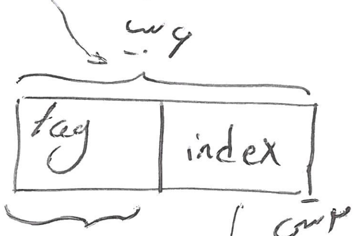


آدرس هایی که در cache داریم تنها ۹ بیت

پس با این روش حافظه ۱۲+۶=۱۸ بیت شد.

مثال: حافظه ۶۴×۱۶ bit خانه
کش ۴ × ۴ خانه

$۶۴ = ۲^{۶}$



پس

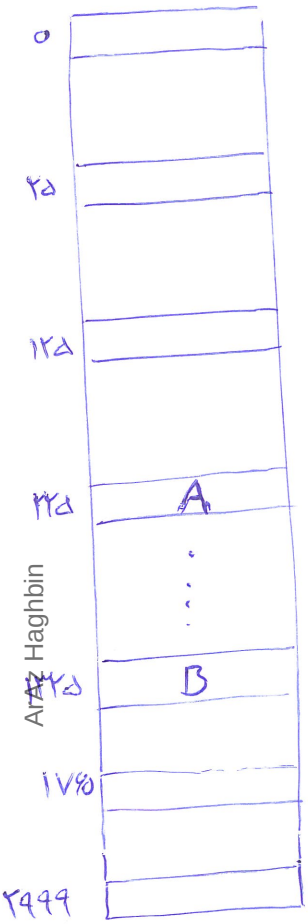
	tag	data
۰۰	۴ bit	۱۶ bit
۰۱		
۱۰		
۱۱		

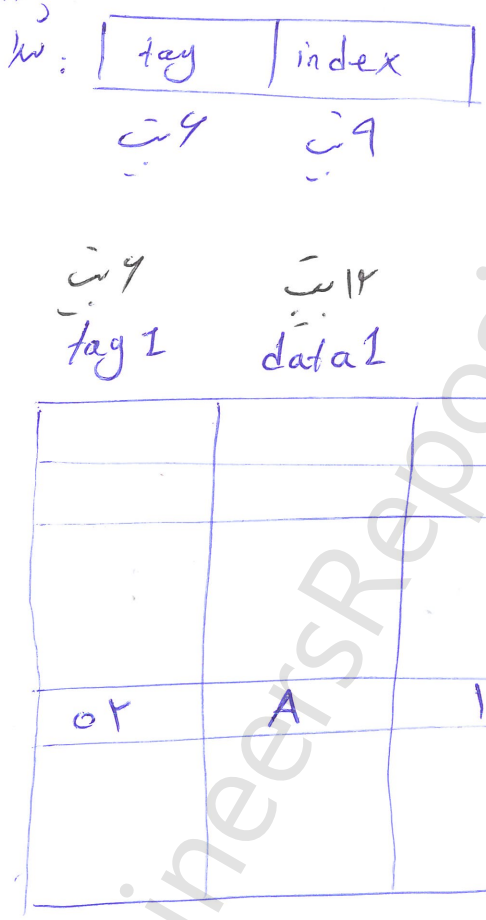
جواب ۴+۱۶ = ۲۰ bit

صفحه ۹ دارد.

روش سوم: Set Associated mapping: نگاشت تداعی مجموعه‌ای:

مثال: Decimal





مثال: حافظه‌ی با ظرفیت ۲۰۴۸ خانه ۱۵ بیتی داریم که میخواهیم بین ۳۲ خانه داده تقسیم کنیم! اندازه‌ی هر خانه بین روش‌های مختلف متغیر (۸ بیت) است.

حافظه: ۲۰۴۸×۱۵

$\downarrow$

$۲^{۱۱}$

$\downarrow$

آدرس دهی حافظه ۱۱ بیتی انجام میشود

بین: ۳۲×۸

$\downarrow$

$۲^{۵}$

$\downarrow$

آدرس دهی بین ۵ بیتی انجام میشود

جواب جلسه بعد!

روش کامل:

شماره آدرس حافظه	data
⋮	⋮

۱۱ + ۱۵ = ۲۶ bits

روش مستقیم:

tag	data
۱۱-۵=۶	⋮

۶ + ۱۵ = ۲۱ bits

روش:

tag 1	Data 1	tag 2	Data 2
۶	۱۵	۶	۱۵
⋮	⋮	⋮	⋮

۶ + ۱۵ + ۶ + ۱۵ = ۴۲ bits