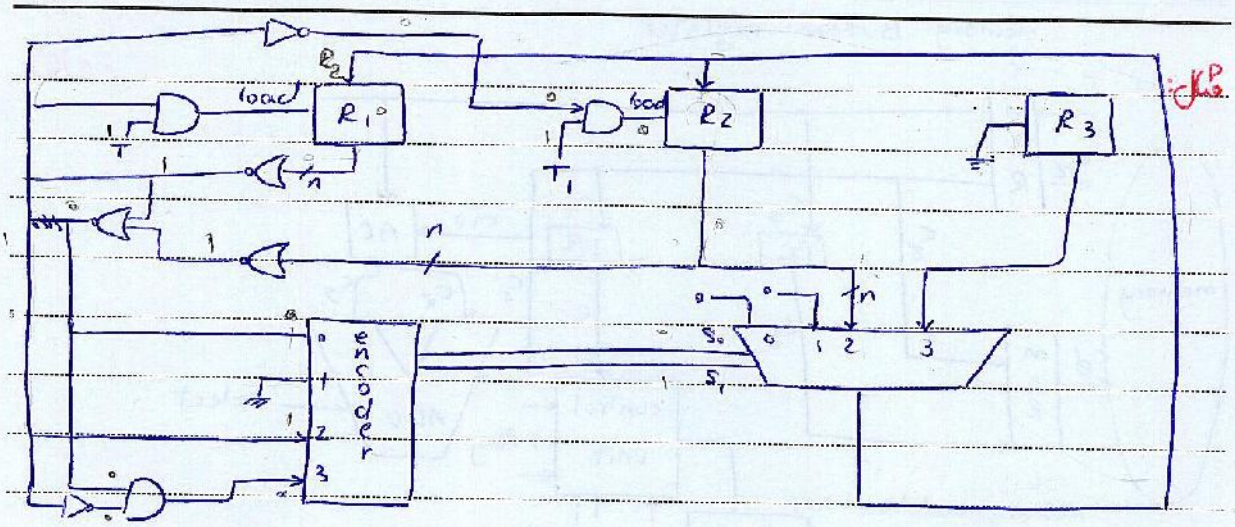
Subject:

Year. Month. Date. ( )

مثال:

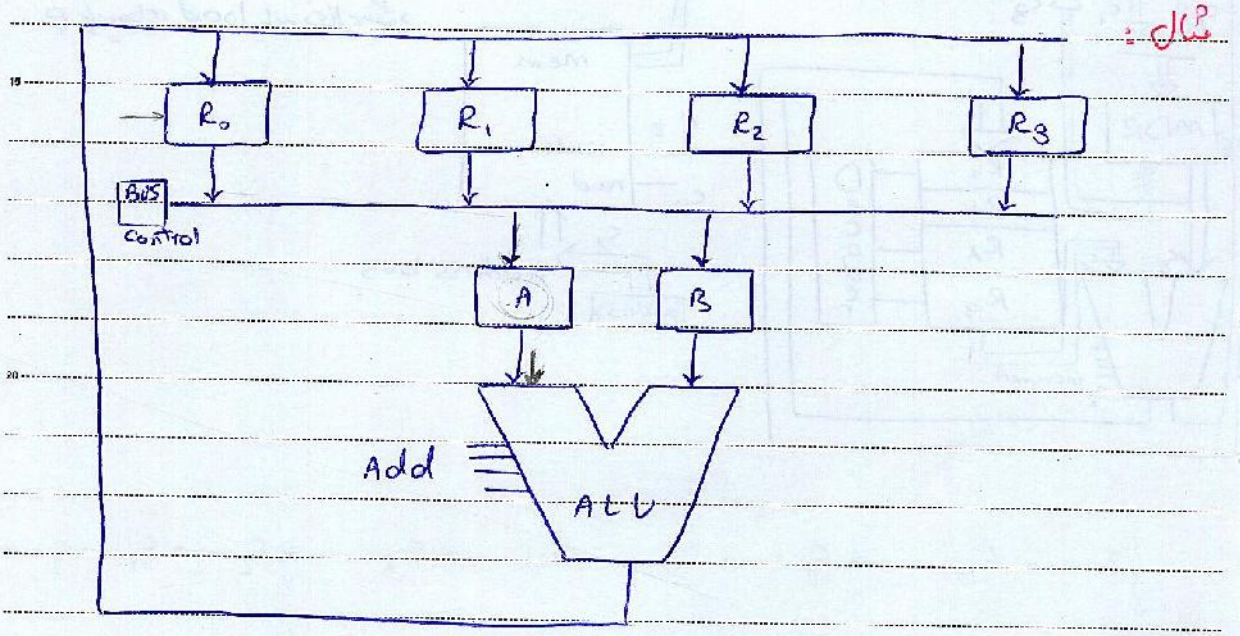


اگر $T_1 = 1$ باشد محتوای R_1 ، R_2 و R_3 را برابر قرار دهید.

if $R_1 = 0$ & $R_2 = 0$ → در صورتی که هر دو صفر باشند

$R_1 \leftarrow R_2$

مثال:



برای انجام عمل $R_1 \leftarrow R_0 + R_1$ ، $R_1 \leftarrow R_2 + R_3$ ضمن clock اضافی دارد.

$R_1 \leftarrow R_0 + R_1$ (3 clock):

$A \leftarrow R_0$ (1)

$B \leftarrow R_1$ (2)

$R_1 \leftarrow add$ (3)

$R_1 \leftarrow R_2 + R_3$ (3 clock):

$A \leftarrow R_2$ (1)

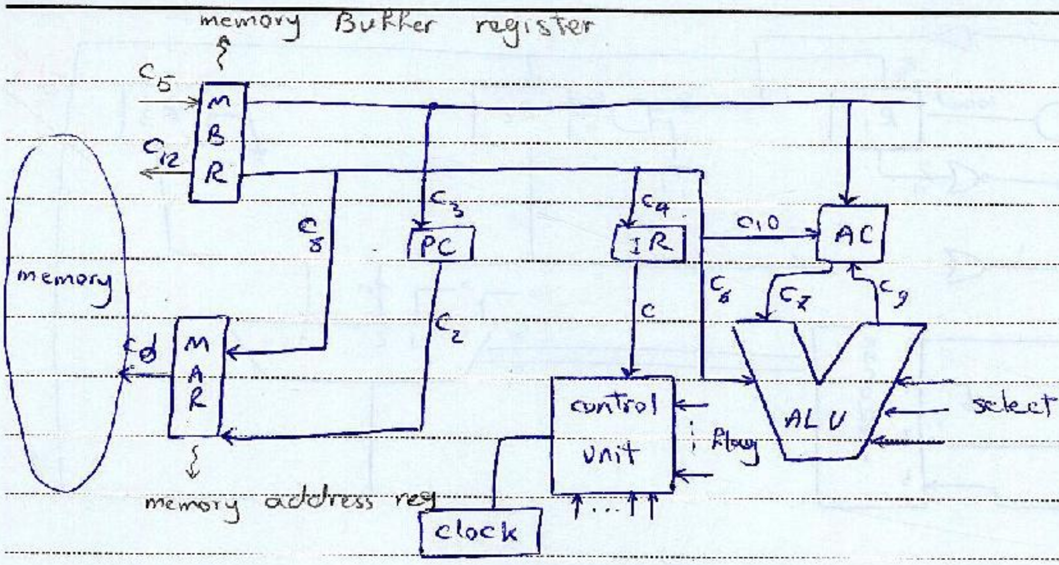
$B \leftarrow R_3$ (2)

$R_1 \leftarrow add$ (3)

← در هر clock یک عمل انجام می‌شود.

Subject:
Year. Month. Date. ( )

مثال:

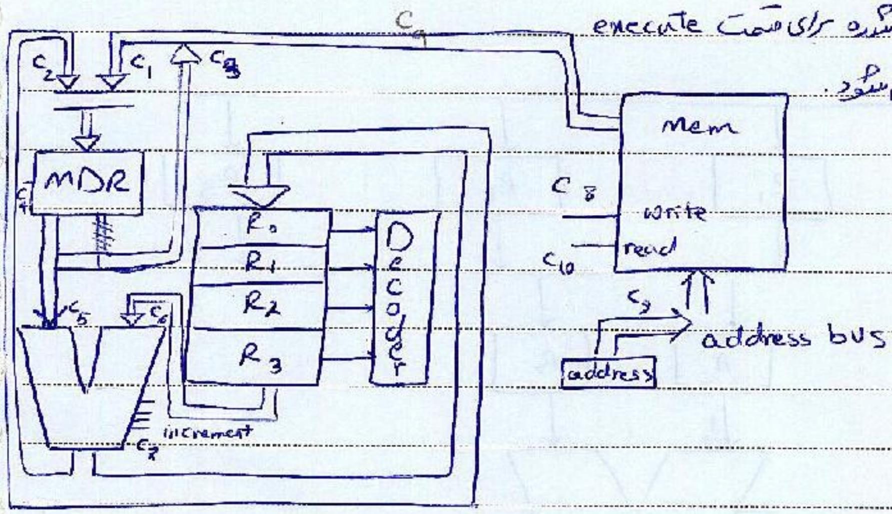


1) AR ← PC C_2

IR ← M[AR] C_0 , C_5

مثال:

دستور increm ، Fetch شده برای قسمت execute
چه بارهای load باید فعال شود.



$C_9 \rightarrow C_{10} \rightarrow C_1 \rightarrow C_4 \rightarrow C_5 \rightarrow C_7 \rightarrow C_2 \rightarrow C_4 \rightarrow C_3 \rightarrow C_8$

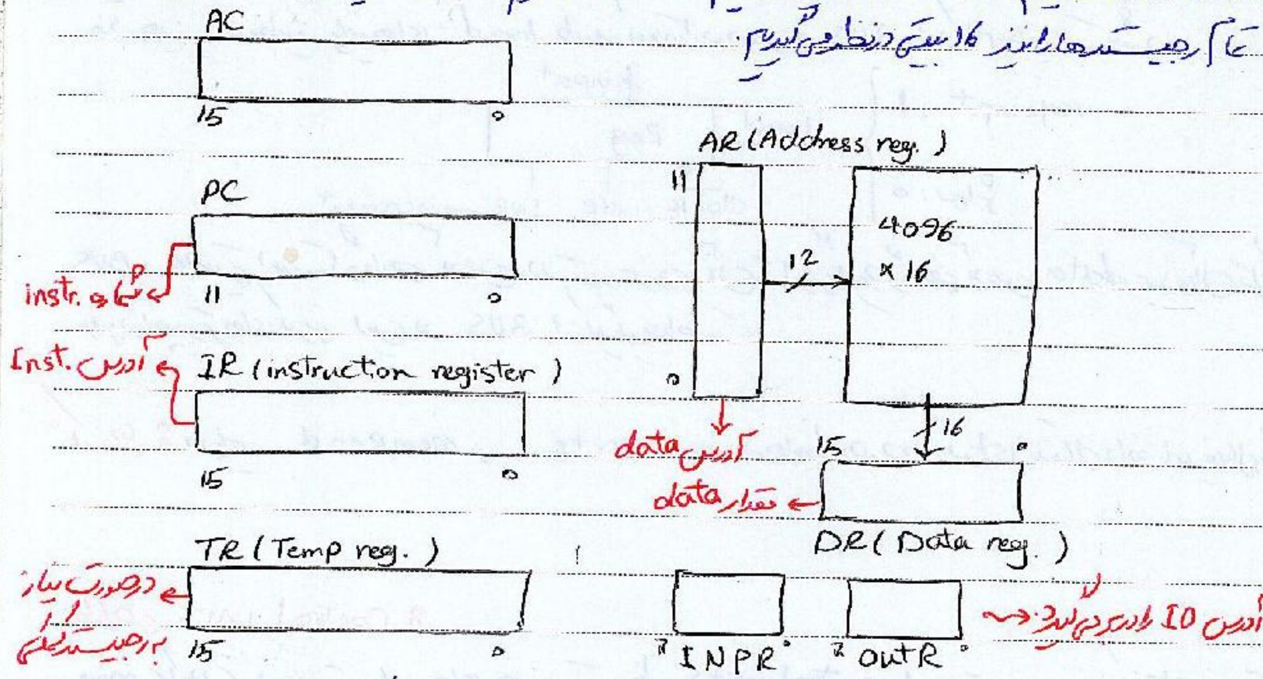
دوره Fetch
گویند

PAPCO

Subject:
Year. Month. Date. ( )

: data path

می‌خواهیم یک پردازنده ۱۶ بیتی از نوع RISC طراحی کنیم که به نوعی شیوه آدرس‌دهی ۱۱ بیت آن مستقیم ۱۲ حافظه‌ای مستقیم ۱۳ حافظه‌ای غیرمستقیم استفاده می‌کنیم. از آن جایی که پردازنده ۱۶ بیتی است تمام رجیسترها باید ۱۶ بیتی در نظر بگیریم



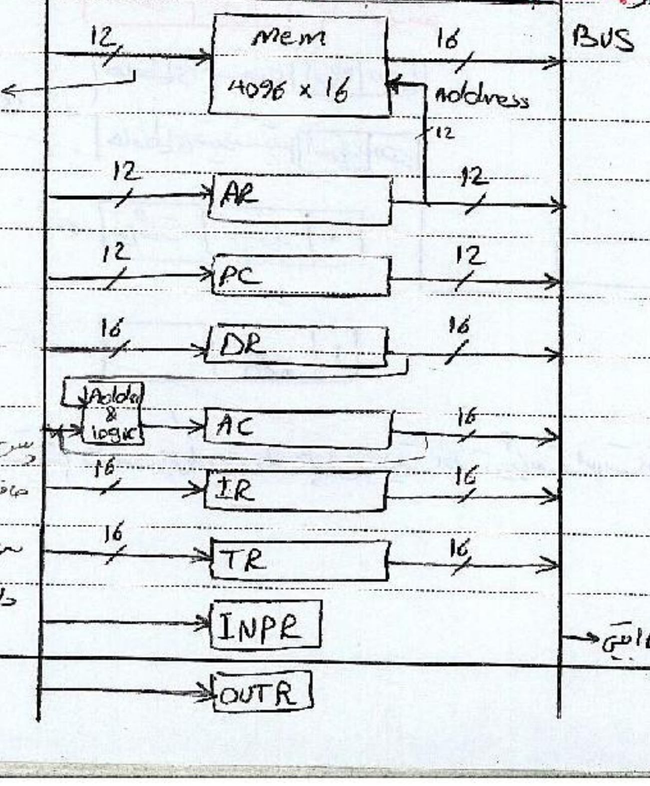
PC و IR به عنوان ... AR و DR هم ... وصل هستند ولی در حافظه فقط ... تنها از آنها مقداردهی شده و خروجی را تعیین می‌کنند (البته در این پردازنده فقط AR را وصل می‌کنیم و در صورت نیاز مقدار PC را به AR منتقل می‌کنیم) MUX

AR, PC: اگر عملیات از نوع bus باشد ۱۲ بیتی می‌شود.

DR, TR, AC

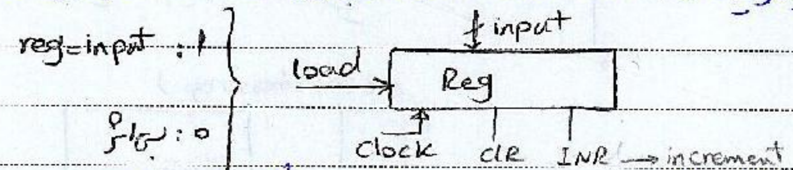
$4096 = 2^{12}$ پس ۱۲ بیت برای ... حافظه احتیاج داریم.

همه رجیسترها باید پین load و clock داشته باشند.



از آن جایی که نباید همه خروجی ها به BUS وصل شوند با قرار دادن یک 3C MUX سلکتور جلوی خروجی این مشکل برطرف می شود.

همه رجیسترها نیز پایه های load دارند که مقدار موجود روی BUS داخل کدام رجیستر شود

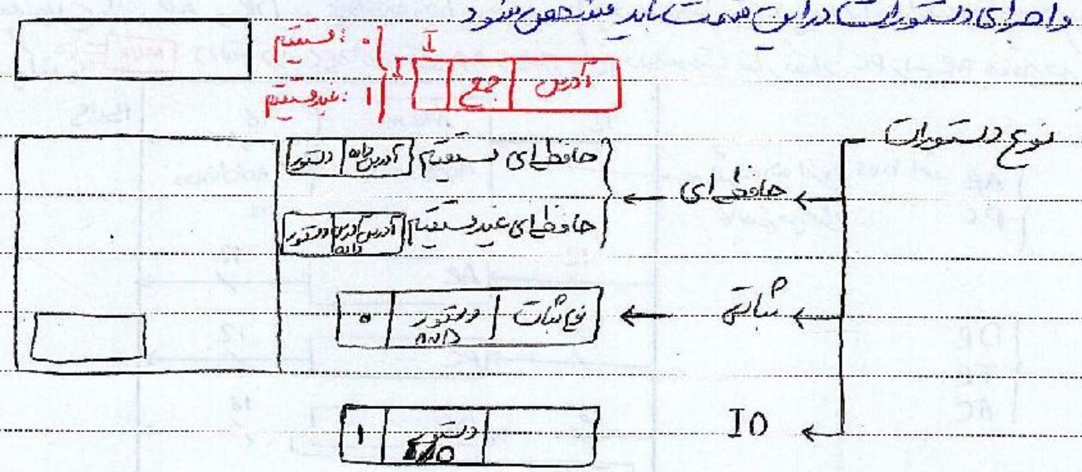


BUS کامپیوتر است که هم آدرس دستور بر روی آن قرار می گیرد و هم آدرس data بر روی آن قرار می گیرد ولی در کامپیوترهای امروزی این دو BUS از یکدیگر جدا هستند

نکته: دو پایه MemRead و memWrite در حافظه وجود دارد که برای انتقال داده باید فعال باشد

طراحی Control unit:

مشخص کردن موقعیت پایه های رجیستر وظیفه control units است و همچنین کنترل عملیات ها و اجرای دستورات در این قسمت باید مشخص شود

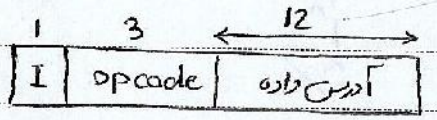


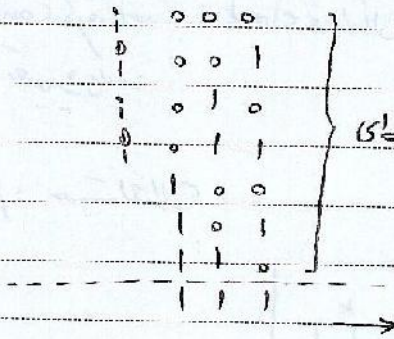
در دستورات ثباتی تنها حالت مستقیم را در نظر می گیریم ولی چون فقط دستور ثابت است مقدار I وجود دارد و برابر صفر است

Subject:

Year. Month. Date. ( )

در حالت کلی داریم:





دستورات ثباتی ← $I = 0$

دستورات I/O ← $I = 1$

تعداد دستورات حافظه‌ای $\longrightarrow 14 = (7 + 7)$ ، $opcode = 0, \ldots, 6$

indirect

direct

تعداد دستورات ثباتی $\longrightarrow$ 12 ، $opcode = 7$ ، $I = 0$

000...01
000...10
100...0

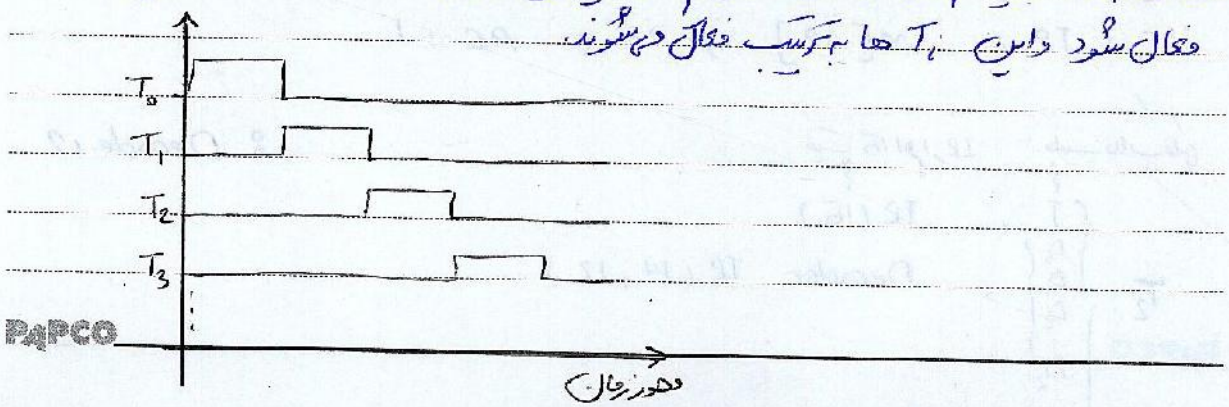
از آن جایی که تعداد رجیسترها محدود است می‌توان از 12 بیت آدرس داده به عنوان opcode استفاده کرد.
در این حالت تعداد معدودی از حالت‌های 12 بیتی را انتخاب می‌کنیم مثلاً 12 بیتی که در هر بار تحلیل آنیت‌ها یک باشند.

تعداد دستورات I/O $\longrightarrow$ ، $opcode = 7$ ، $I = 1$

از 12 بیت نیز می‌توان برای تعیین آدرس port استفاده کرد.

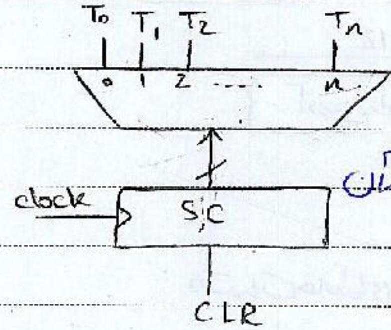
طراحی sequencer (توالی‌گر)

sequencer باید گام‌های مختلف الگوریتم فون نیومن را ایجاد کند و در هر لحظه در زمان یکی از T_iها فعال شود. این T_iها به ترتیب فعال می‌شوند.



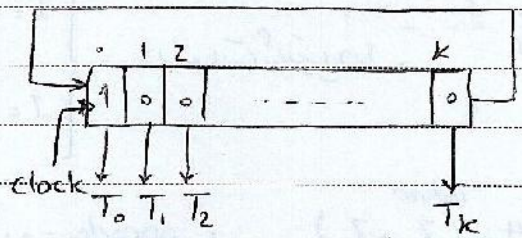
Subject:
Year. Month. Date. ( )

روش اول:



یک counter که متناسب با clock مقدار آن از ۰ تا n تغییر می‌کند

روش دوم: بیت لغزان



از یک شیفت رجیستر بزرگ استفاده می‌کنیم با این شرط که در هر لحظه تنها یکی از رجیسترها مقدار یک را دارد

طراحی الگوریتم فون نیومن:

مراحل مختلف این الگوریتم به صورت زیر ساده سازی می‌شود

1) Fetch: در این طراحی از مسیر A استفاده می‌کنیم یعنی قالب دستورات به صورت زیر است:

شرط : R ← ...

$T_0 : AR \leftarrow PC$

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

2) Decode:

(بیت ۱۵ ام از IR)

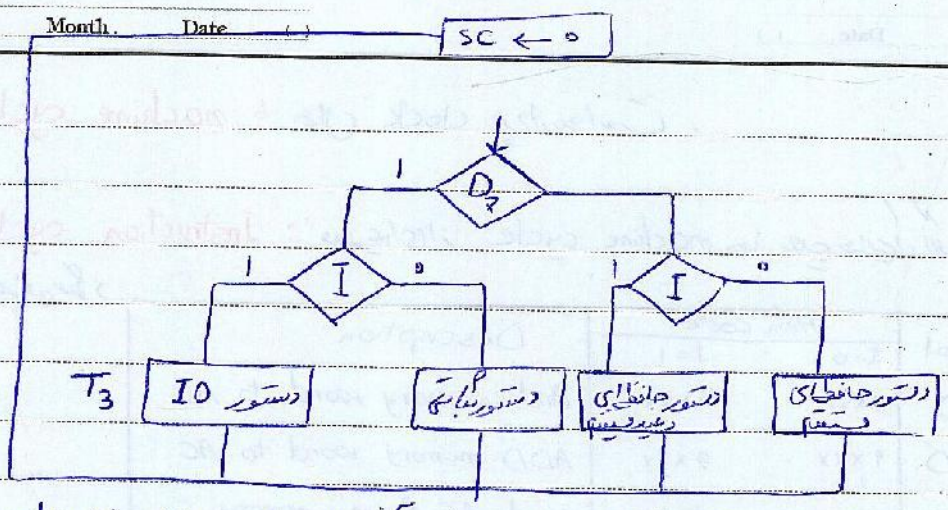
$T_2 :$
- $I \leftarrow IR(15)$
- $D_0, D_1, D_2, \dots, D_7 \leftarrow$ Decoder $IR(14 - 12)$
- $AR \leftarrow IR(11 - 0)$

PAPCO

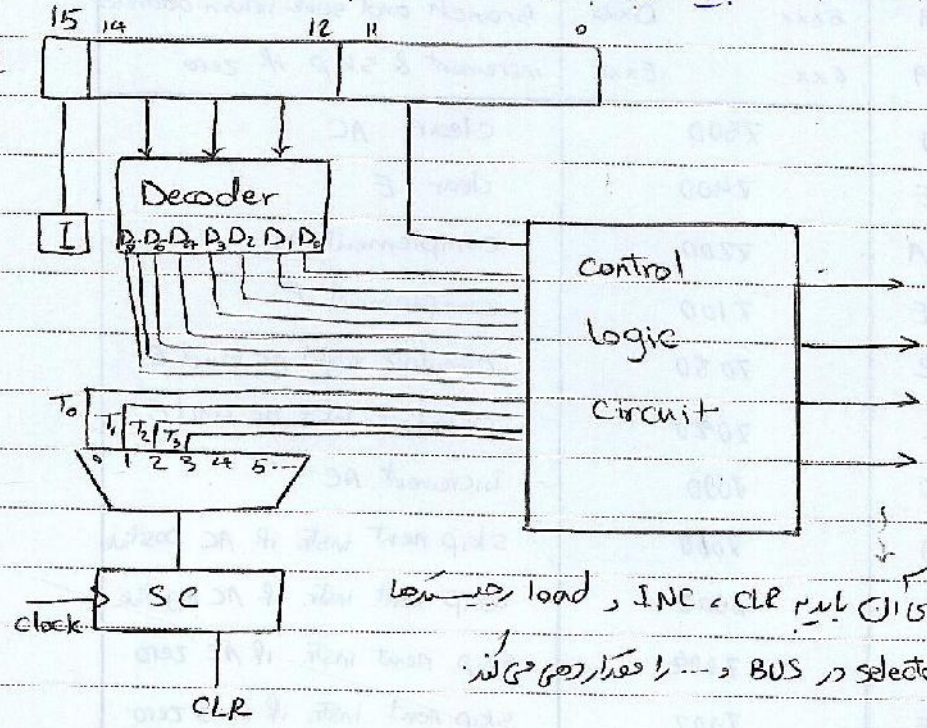
Subject :

Year . Month . Date

13



شروع مرحله اجرای این دستورات T_3 است ولی چون بعضی از آنها باید برای هم برای خواندن operand مصرف کنند بیشتر از یک clock نیاز دارد. از میان این دستورات مبادله و IO نیازی به خواندن operand ندارد زیرا در opcode 12 بیت مربوط به آن استفاده از ثبات یا شماره پورت مشخص می‌شود



خروجی‌های آن باید INC ، CLR و load رجیسترها

پایه‌های selector در BUS و ... را مقداردهی می‌کند.

برای سادگی فرض کرده‌ایم که تنها یک دستگاه ورودی (کیبورد) و یک دستگاه خروجی (مانیتور) وجود دارد پس شماره پورت نداریم.

machine cycle : هر clock پردازنده است.

Instruction cycle : مجموعه‌ای از machine cycle ها یعنی زمانی که الگوریتم از بالا به پایین اجرا می‌شود

دستورات حافظه‌ای

Symbol	Hex code I=0	I=1	Description
AND	0xxx	8xxx	And memory word to AC
ADD	1xxx	9xxx	ADD memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISA	6xx	Exxx	increment & skip if zero

دستورات محاسباتی

Symbol	Hex code	Description
CLA	7800	clear AC
CLE	7400	clear E
CMA	7200	complement AC
CME	7100	complement E
CIR	7080	Circulate right AC and E
CIL	7040	Circulate left AC and E
INC	7020	Increment AC
SPA	7010	skip nent instr. if AC postive
SNA	7008	skip nent instr. if AC negetive
SZA	7004	skip nent instr. if AC zero
SZE	7002	skip nent instr. if E is zero
HLT	7001	Halt computer

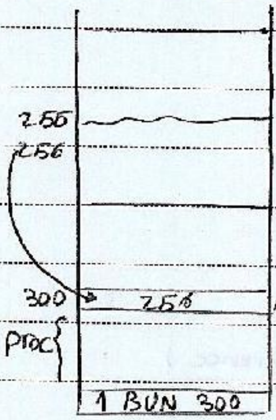
I/O

Symbol	Hex code	Description
INP	F800	input charater to AC
OUT	F400	output character from AC
SKI	F200	skip on input flag
SKO	F100	skip on output flag
ION	F080	interrupt on
IOF	F040	Interrupt off

چون تک‌تک باید در تمام IO ها [illegible]
داریم نیاز به [illegible]

BUN : به آدرسی که در مقابل آن نوشته می‌شود jump می‌کند

BSA : آدرس محل جاری را نگه می‌دارد و پس از jump کردن می‌تواند به خانه اصلی برگردد. برای اینکه اولین خط proc مربوط به آدرس بازگشت اختصاص می‌دهد و زمانی که تابع به طور کامل اجرا شد آدرس خانه جاری را از اولین خط proc برمی‌دارد



1 BUN xyz ← به صورت indirect پرش می‌کند یعنی آدرس اصلی را از خانه xyz بخوان و به آن جا jump کن.

0 BUN xyz ← به صورت direct آدرس را خوانده و jump می‌کند.

HLt : پردازنده به حالت تعلیق می‌رود و هیچ دستوری اجرا نمی‌شود. البته پردازنده فعال است. باید reset شود تا دوباره به راه بیفتد.

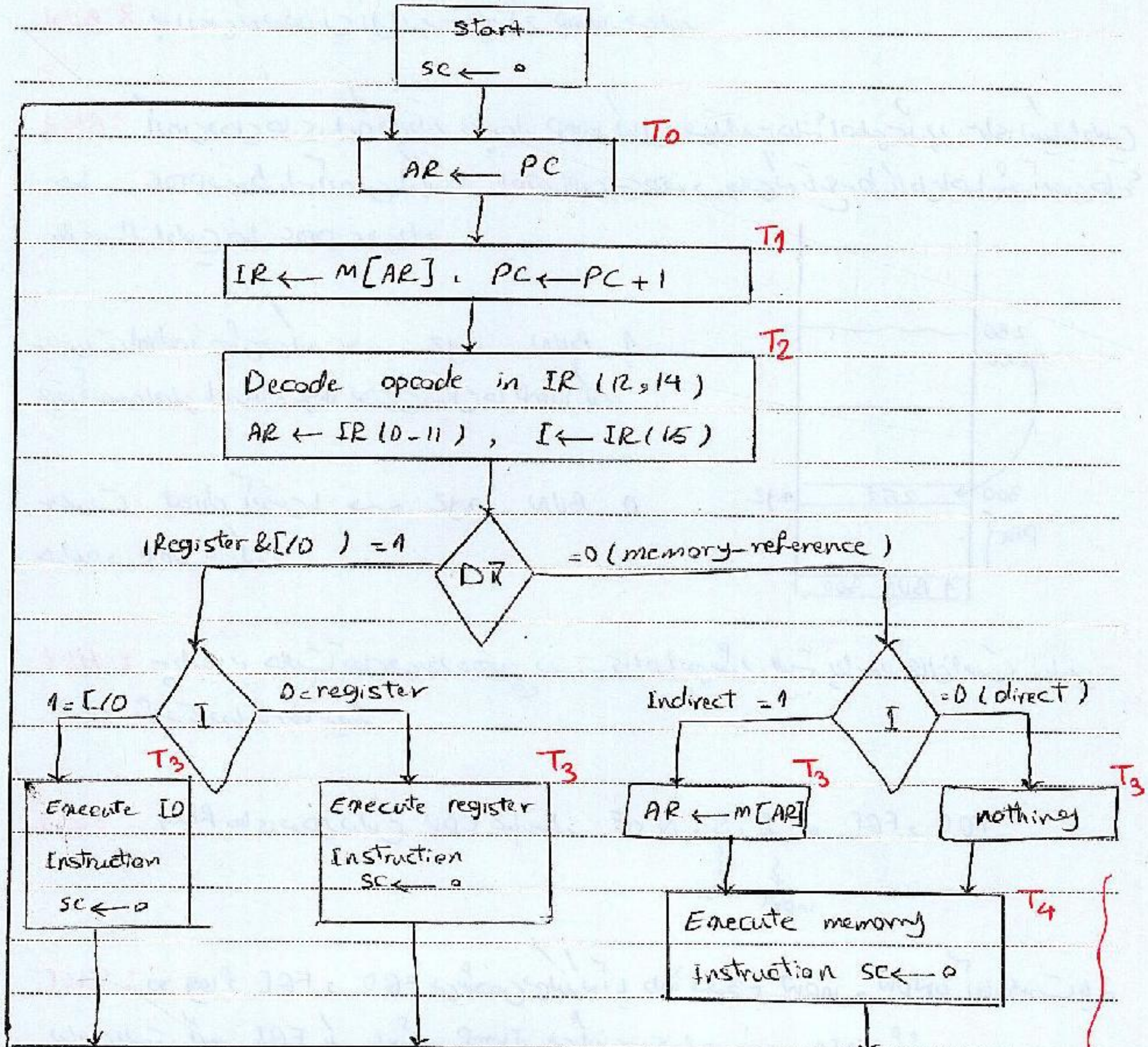
نکته : flag های موجود در CPU عبارتند از E، N، S، I و FGI، FGO
(I: input، S: sign)

SKI : دو flag FGI و FGO مشخص می‌کنند آیا داده مربوط به input و output آماده است یا نه. در این دستور اگر FGI یک باشد jump می‌کند به چه جایی ... دارد ؟؟

SKO : اگر FGO یک باشد jump می‌کند ؟؟

PAPCO

Subject.
Year. Month. Date. ( )



علت nothing این است که هر دو حالت direct و indirect با یکدیگر تمام شوند و مجبور به طراحی دو مدار جدا از هم نباشیم و مدار پایه‌های load آن‌ها خیلی ساده‌تر می‌شود ولی تاخیر بیشتر است.

$D_7' I T_3$: $AR \leftarrow M[AR]$ ⇝ $D_7 = 0$, $I = 1$, $T_3 = 1$

$D_7' I' T_3$: nothing

$D_7 I' T_3$: Execute a register-reference instr.

$D_7 I T_3$: Execute an input-output instr.

Subject:

Year. Month. Date. ( )

$r = D_7 I' T_3 \Rightarrow$ Register – Reference Instruction

$B_i = IR(i) \quad i = 0, 1, 2, \ldots, 11$

دستورات رجیستری به صورت زیر با شرط های داده شده اجرا می شوند:

	$r:$	$SC \leftarrow 0$
CLA	$rB_{11}:$	$AC \leftarrow 0$
CLE	$rB_{10}:$	$E \leftarrow 0$
CMA	$rB_9:$	$AC \leftarrow AC'$
CME	$rB_8:$	$E \leftarrow E'$
CIR	$rB_7:$	$AC \leftarrow shr\ AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$
CIL	$rB_6:$	$AC \leftarrow shl\ AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$
INC	$rB_5:$	$AC \leftarrow AC + 1$
SPA	$rB_4:$	$if(AC(15) = 0)\ then\ (PC \leftarrow PC + 1)$
SNA	$rB_3:$	$if(AC(15) = 1)\ then\ (PC \leftarrow PC + 1)$
SZA	$rB_2:$	$if(AC = 0)\ then\ (PC \leftarrow PC + 1)$
SZE	$rB_1:$	$if(E = 0)\ then\ (PC \leftarrow PC + 1)$
HLT	$rB_0:$	$S \leftarrow 0$ (S is a start-stop flip-flop)

هر دو شرط باهم برقرار باشد

بصورت برقراری شرط مربوط PC دستها جلو می رود چون در حالت طبیعی یکی PC اضافه می شود.

دستورات حافظه ای به صورت زیر فعال می شوند در اینجا صفر بودن I لازم نیست زیرا Iها خروجی decoder هستند که اگر یکی از D_i ها فعال باشد D_7 حتماً صفر است. مقدار I نیز چک نمی شود زیرا در هر دو حالت direct و indirect خروجی دو جمع آخر AR وارد شده است.

نکته: دستورات زیر حتماً نمی توانند انجام شوند زیرا operand های ورودی ALU است DR و AC است

سپس باید داده مربوط ابتدا وارد DR شود و سپس عملیات ALU انجام شود.

And:

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

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

ADD

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

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

می توانند انجام شوند

Subject:

Year. Month. Date. ( )

AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow Cout$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] + 1$, if $[M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

Load to AC :

D_2T_4 : $DR \leftarrow M[AR]$

D_2T_5 : $AC \leftarrow DR$, $SC \leftarrow 0$ } این دو دستور نمی‌توانند همزمان اجرا شوند زیرا انتقال داده در هر دو از طریق BUS انجام می‌گردد.

STA : store AC

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

BUN : Branch unconditionally

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

BSA : Branch & save Return Address

D_5T_4 : $M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$ → هر دو میکرو دستور BUS را نیاز دارند برای همین نمی‌توانند با هم اجرا شوند.

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

ISZ : Increment and skip if-zero

D_6T_4 : $DR \leftarrow M[AR]$

D_6T_5 : $DR \leftarrow DR + 1$

D_6T_6 : $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

Subject:

Year. Month. Date. ( )

برای طراحی پایه load مربوط به DR می‌توان تمام زمان هایی که DR مقداردهی می‌شود یعنی D_2T_4 و با گیت OR کرد

پیاده‌سازی پایه‌های AR:

همانطور که گفتیم AR در کل مقداردهی می‌شود را مشخص می‌کنیم می‌توان مقدار load، INR، CLR را مشخص کرد

Interrupt $R'T_0$: $AR \leftarrow PC$ LD(AR)

$R'T_2$: $AR \leftarrow IR(0-11)$ LD(AR)

D'_7IT_3 : $AR \leftarrow M[AR]$ LD(AR)

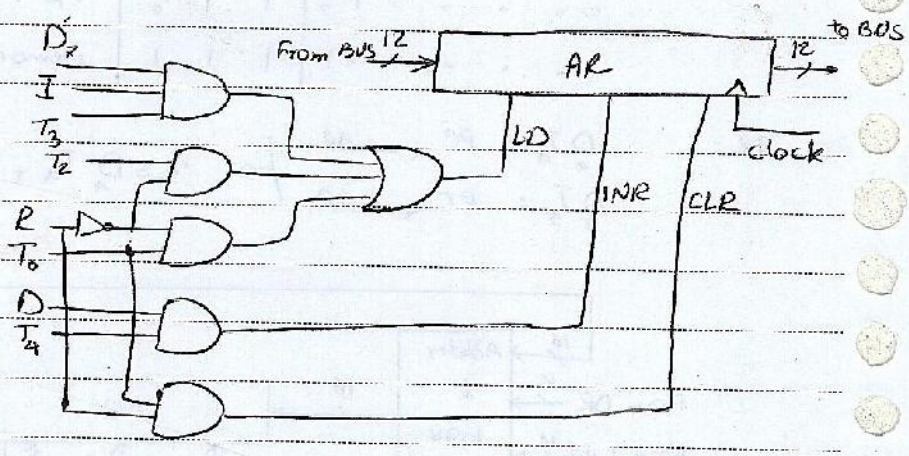
RT_0 : $AR \leftarrow 0$ CLR(AR)

D_5T_4 : $AR \leftarrow AR+1$ INR(AR)

$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$

$CLR(AR) = RT_0$

$INR(AR) = D_5T_4$

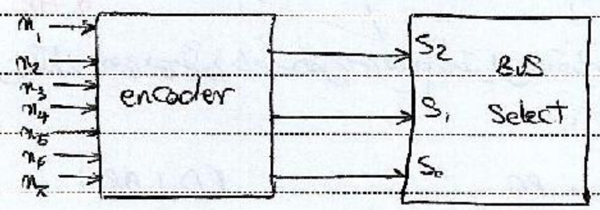


PAPCO

Subject:
Year. Month. Date. ( )

Control of Common BUS

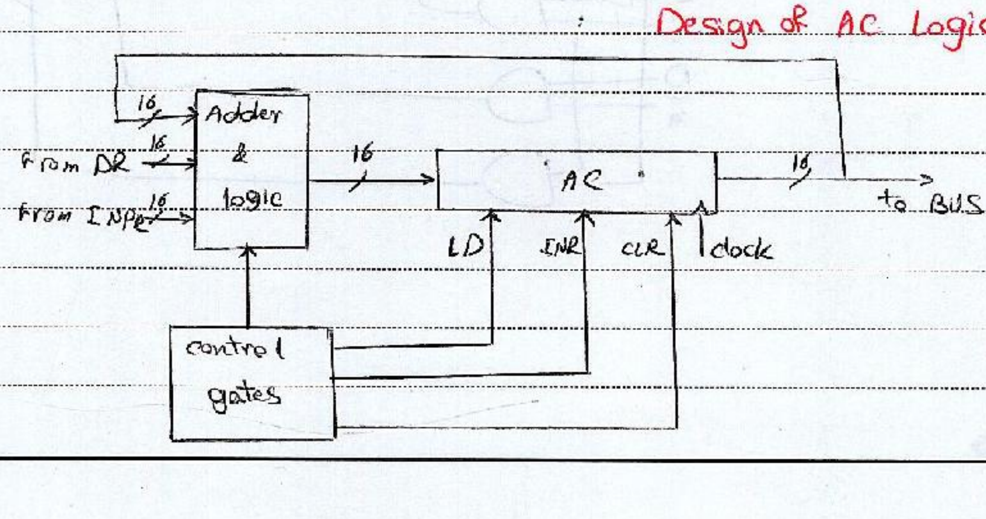
در اینجا به هر یک از رجیسترهای موجود یک عدد نسبت می‌دهیم و سپس همه‌ی دستوراتی که به مقدار آن رجیستر احتیاج دارند را انتخاب می‌کنیم و با یکدیگر OR می‌کنیم.



x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_2	S_1	S_0	selected register
0	0	0	0	0	0	0	0	0	0	none
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
[illegible]	0	0	0	0	0	1	1	1	1	memory

AR: $D_4 T_4$: PC ← AR
$D_5 T_5$: PC ← AR
$\Rightarrow x_1 = D_4 T_4 + D_5 T_5$

Design of AC Logic



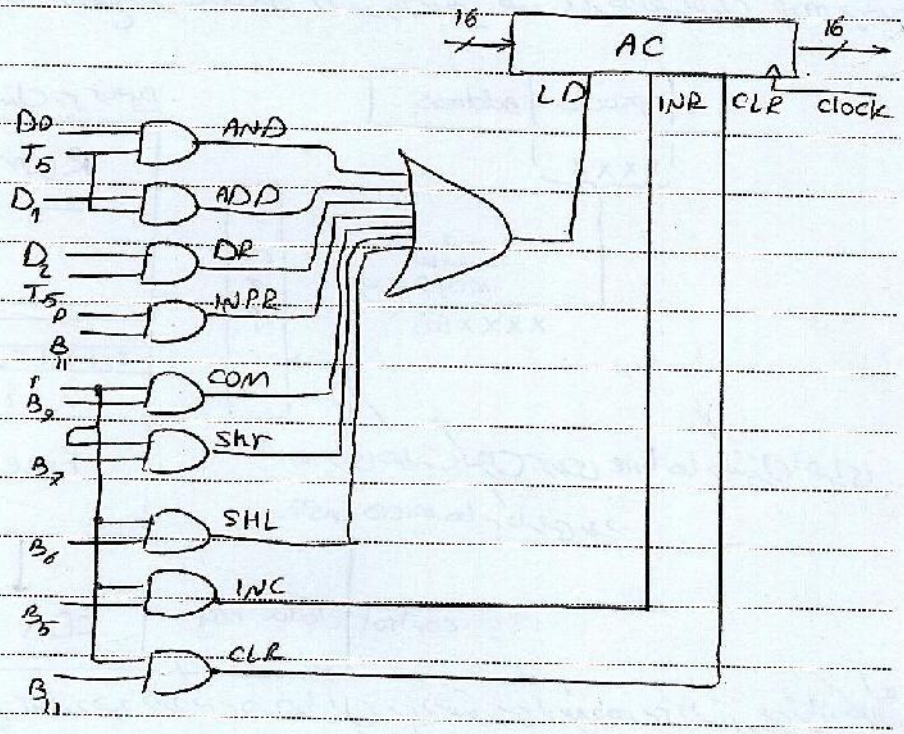
Subject:

Year. Month. Date. ( )

حالا دستوراتی که مقدار AC را تغییر می دهند بیان کنید:

D_0T_5	$AC \leftarrow AC \wedge DR$
D_1T_5	$AC \leftarrow AC + DR$
D_2T_5	$AC \leftarrow DR$
PB_{11}	$AC(0..7) \leftarrow INPR$
rB_9	$AC \leftarrow AC'$
rB_8	$AC \leftarrow shr\,AC,\ AC(15) \leftarrow E$
rB_7	$AC \leftarrow shl\,AC,\ AC(0) \leftarrow E$
rB_{11}	$AC \leftarrow 0$
rB_5	$AC \leftarrow AC + 1$

در نتیجه با توجه به شرط های این دستورات می توان پایه های AC را مشخص کرد:



Subject:
Year. Month. Date. ( )

با توجه به اینکه انجام عملیات در زمان های متفاوتی انجام می‌شود باید طول کلاک را برابر با بزرگترین مقدار در نظر بگیریم

طراحی واحد کنترل

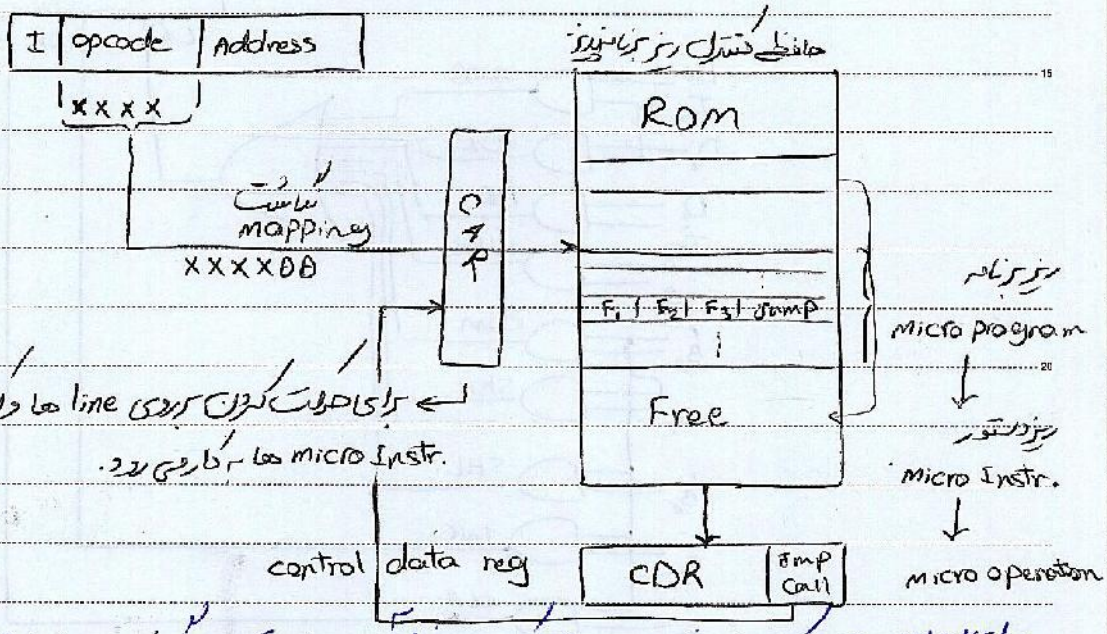
hard wires سیم بندی شده : طراحی واحد کنترل به صورت سخت افزاری و بصورت کردن سیم ها انجام می‌شود.

micro programmed : ریز برنامه پذیر : هر واحد کنترل درون یک block قرار می‌گیرد و قابل کنترل به برنامه ریزی است.

طراحی واحد کنترل ریز برنامه پذیر : Micro programmed control Design

در این شرایط نیازی به تغییر سخت افزار نیست و در صورت نیاز به اعمال تغییر کافی است تنها برنامه آن را عوض کنیم. واحد کنترل در این روش قابل program است

برای اشاره به یک خانه کنترل برنامه ریز تعریف می‌کنیم و program کردن ROM وجود در این قسمت دستورات اجرایی می‌شوند. به ازای هر opcode دستور را به یک قسمت از خانه کنترل map می‌کنیم



برای اجرای دستور یک ریز برنامه وجود دارد که به هر خط آن ریز دستور می‌گویند و هر ریز دستور می‌تواند چند micro operation داشته باشد. F1 و F2 و F3 به صورت موازی با یکدیگر اجرا می‌شوند.

Subject:
Year. Month. Date. ( )

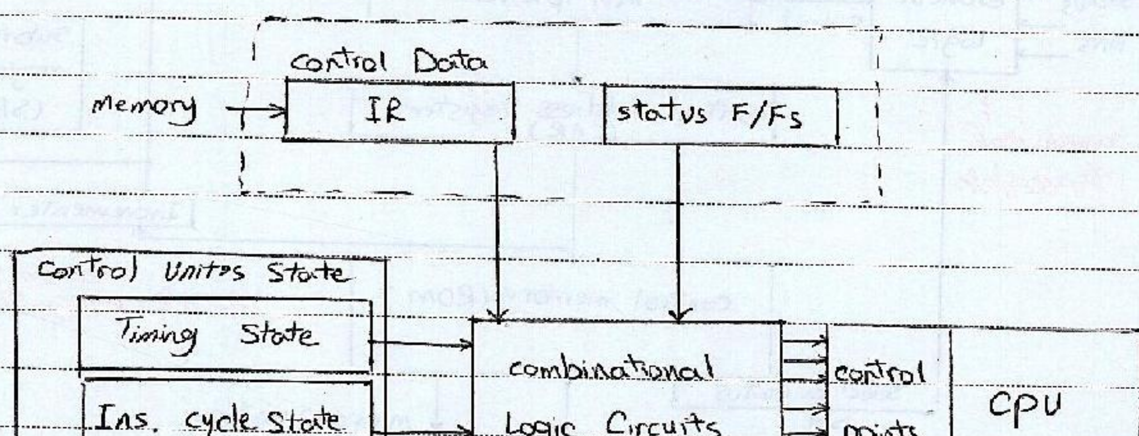
درنتیجه در این روش نیازی به سیم بندی مجدد نداریم. کافی است با عوض کردن برنامه ROM شیوه اجرای دستورات را تغییر داد.

شیوه ثابت باید به صورت ساده و سریع انجام شود. یکی از شیوه های طراحی این است که در صورت 4 بیتی بودن opcode تنها با قرار دادن دو صفر در مقابل آن دستور را در حافظه کنترل ثابت کنیم یعنی به هر opcode ، ٤ line اختصاص می دهیم و در صورتی که جا نشد یک فضای خالی برای آن در نظر می گیریم و در صورت نیاز به آن ها jump می کنیم. ولی با 6 بیت آدرس دهی نمی توان فضای خالی برای آن تقسیم کرد. درنتیجه حافظه را 128 خطی در نظر می گیریم و چون برای آدرس دهی به 7 بیت احتیاج داریم سپس آدرس را به صورت زیر تعریف می کنیم

0xxxx00 ⟶ بخش بالایی حافظه map می شود

1xxxx ⟶ فضای خالی در انتهای حافظه

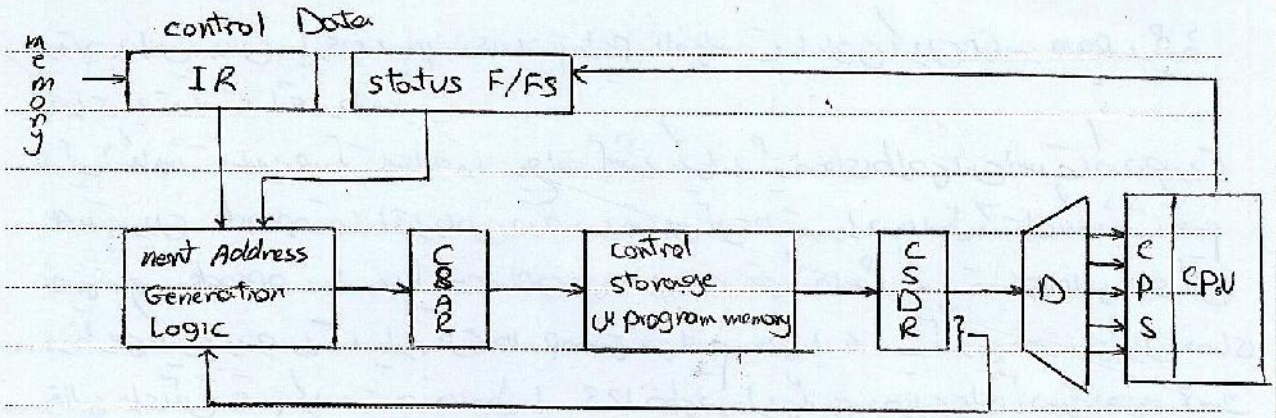
نکته مهم 8 بار خود دانش Fetch و decode ، opcode ندارند ولی در حافظه کنترل در قسمت free یک micro prog. برای آن تعریف می کنیم



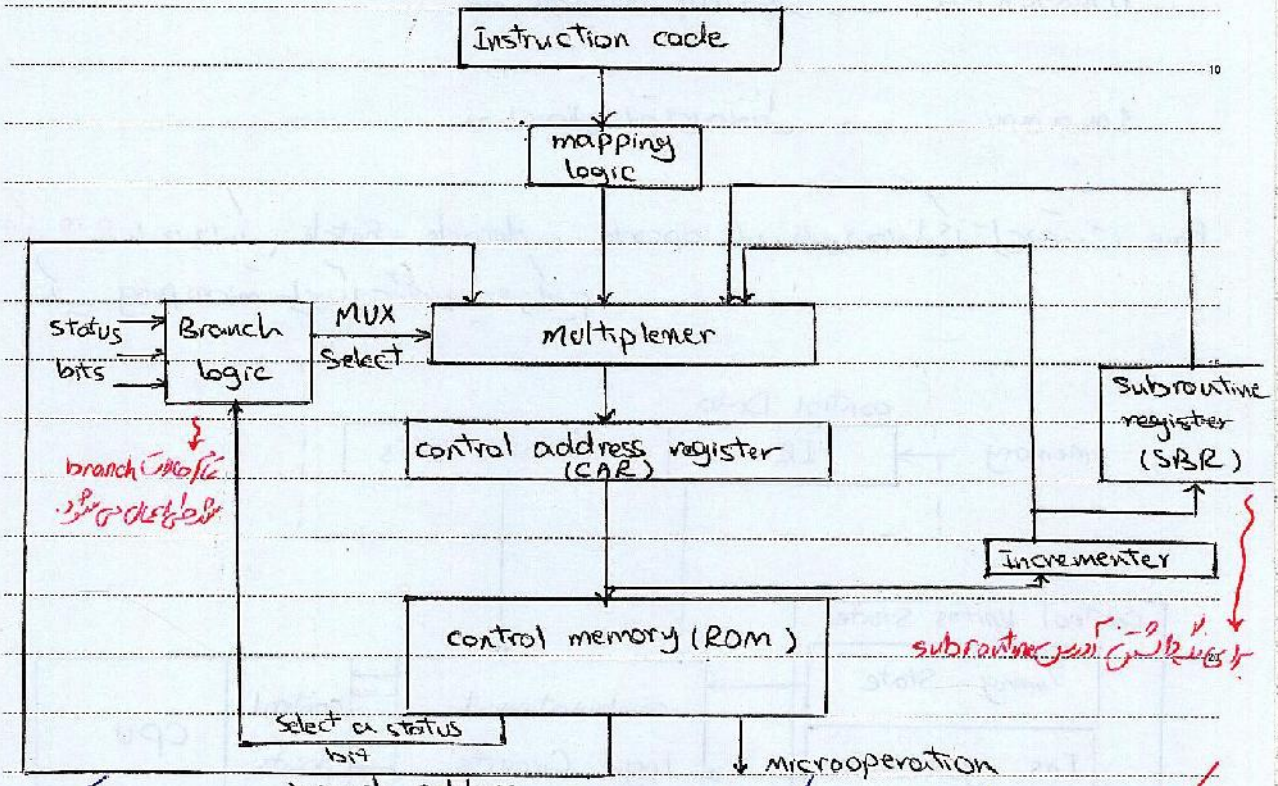
combinatenal Logic Circuit (hard wired)

Subject:

Year. Month. Date. ( )



Micro program



نکته: در این روش تعداد زیادی توان یک call داشت در صورت وجود call های تودرتو آدرس ها گم می شود. یکی از موارد استفاده از call این است که یک دستور Indirect اجرا می شود دیگر نیاز دارد یک بار دیگر از حافظه بخواند که این عمل را با call می توان انجام داد که امکان rat شدن به اشتبا را دارد.

Subject:

Year. Month. Date. ( )

یکی از نحوه های دیگر mapping این است که مقدار 010 را جلوی opcode قرار دهیم.

opcode of Instructions

ADD 0000

AND 0001

LDA 0010

STA 0011

BUN 0100

mapping bits 10 [xxxx] 010

10 [0000] 010 → ADD Routine

10 [0001] 010 → AND Routine

10 [0010] 010 → LDA Routine

10 [0011] 010 → STA Routine

10 [0100] 010 → BUN Routine

مثال: قالب دستورات در حافظه بصورت زیر است. دستورات و کدهای تعریف شده عبارتند

seg برای آنها اختصاص یافته و دارای 4 خط دستور هستند.

۴	۳	۳	۲	۲	۷
F1	F2	F3	CD	BR	AD

هر line، 20 بیت دارد

F1, F2, F3: micro operation field

CD: شرط branching

BR: branching حالت 3 (jump, return, call)

AD: آدرسی که در آن branch می‌شود / در صورت حافظه کنترل است و 12 بیت دارد

Symbol	opcode	Description
ADD	0000	AC ← AC + M[EA]
Branch	0001	if (AC<0) then (PC ← EA)
Store	0010	M[EA] ← AC
Exchange	0011	AC ← M[EA], M[EA] ← AC

EA is the effective address

Subject:

Year. Month. Date. ( )

از آن جایی که سه operation F1 ، F2 و F3 می‌توانند به صورت موازی اجرا شوند دستورات را به گونه‌ای بین این سه تقسیم می‌کنیم که امکان انجام موازی دستورات وجود داشته باشد.

F1		
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0-10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	write

F2		
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	Read
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0-10) \leftarrow PC$	PCTDR

F_3		
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow AC'$	COM
011	$AC \leftarrow shl\ AC$	SHL
100	$AC \leftarrow shr\ AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

اگر از یک F استفاده نمی‌کنیم مقدار none را در آن قرار می‌دهیم.

مقادیری که به عنوان شرط می‌توانند قرار بگیرند و همین طور حالت‌های مختلف branch در جدول‌های زیر وجود دارد.

Subject:
Year. Month. Date. ( )

CD	condition	Symbol	comments
00	Always = 1	U	unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR		
00	Jmp	CAR ← AD if condition = 1 CAR ← CAR+1 if condition = 0
01	Call	CAR ← AD, SBR ← CAR+1 if condition = 1 CAR ← CAR+1 if condition = 0
10	Ret	CAR ← SBR (return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

دستور map عمل map کردن یک قالب دستور درون حافظه کنترلی را انجام می دهد این دستور برای الگوریتم Fetch به کار می رود.
از دستور call نیز می توان برای آن قسمت از subroutine ها یا زیربرنامه ها که مشترک هستند استفاده کرد. مثلاً اگر یک قسمت از یک روتین با دیگری برابر باشد نیازی به نوشتن دوباره ی دستورات نیست و با یک call به آن محل و بازگشت می توان این کار را انجام داد.

نکته ی ۲: از micro program ها در کامپیوترهای CISC استفاده می کنیم.
از معایب این روش سرعت کمتر و پیچیدگی سخت افزاری بیشتر است. از طرفی تعداد صفحه های ذخیره شده در این روش بسیار زیاد است.
در این تکنیک ها مانند این است که یک کامپیوتر درون کامپیوتر دیگر قرار دارد.

PAPCO

INDRCT:	Read	U	Jmp	Next
	DRTAR	U	Ret	

Subject:

Year. Month. Date. ( )

درون حافظه دستورات بصورت زیر قرار می‌گیرند که word 64 اول مربوط به دستورات ماشین و word 64 دوم برای داده‌های دستوراتی مانند Fetch و ... استفاده می‌شود

Label	MicroOps	CD	BR	AD
ADD:	ORG 0			
	NOP	I	Call	INDRCT
	Read	U	Jmp	Next
	ADD	U	Jmp	Fetch
	ORG 4			
Branch:	NOP	S	Jmp	over
	NOP	U	Jmp	Fetch
over:	NOP	I	Call	INDRCT
	ARTPC	U	Jmp	Fetch
	ORG 8			
STORE:	NOP	I	Call	INDRCT
	ACTDR	U	Jmp	NEXT
	Write	U	Jmp	FeTch
	ORG 12			
Exchange:	NOP	I	Call	INDRCT
	Read	U	Jmp	Next
	ACTDR, DRTAC	U	Jmp	Next
	Write	U	Jmp	Fetch
	ORG 64			
Fetch:	PCTAR	U	Jmp	next
	Read, INCPC	U	Jmp	next
	DRTAR	U	Map	Decode

Fetch data

Fetch Data

Fetch data

Fetch data

اگر دستور باشد خواندن data

Subject:

Year. Month. Date. ( )

سرعت سخت افزار hard wire بیشتر است ولی اجرای دستور فعلی است باید بیشتر نباشد و

مشکل به اندازهٔ clock در هر دو دارد.

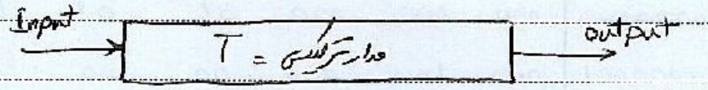
باید تعداد F ها را به اندازه‌ای قرار دهیم که برابر حداکثر تعداد دستوراتی که به صورت موازی اجرا می‌شود باشد.

	decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
Enchange	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
Fetch	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
ENDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

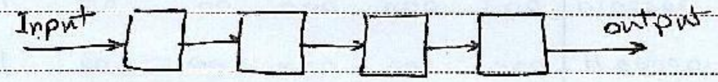
Subject:
Year. Month. Date. ( )

pipeline : (خط لوله)

برای افزایش سرعت به کار می رود. بدون پایپ‌لاین مدار ترکیبی بزرگی طراحی شده که با گرفتن ورودی های داده، عملیات طولانی روی آن ها خروجی آماده می شود. می توان این عملیات را split کرد و به عملیات کوچکتر تقسیم کنیم.



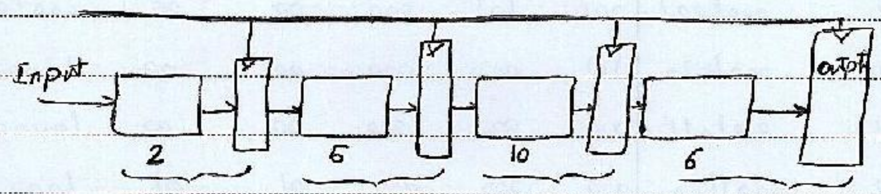
تاخیر $= nT$



تاخیر $= kT_p + (n-1)T_p$

(kT_p: اولین داده — $(n-1)T_p$: n داده — تفاوت T ها اختلاف دارند)

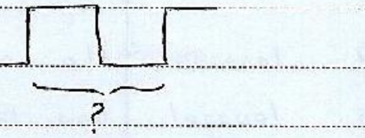
در حالت دوم زمانی که عملیات اولین بخش انجام شد بلا استفاده می ماند در نتیجه می توان از آن برای ورودی های بعدی استفاده کرد و چندین دستور همزمان اجرا می شود. برای جلوگیری از glitch ها و تداخل محاسبات قسمت های مختلف از تعدادی رجیستر یا latch برای نگهداری نتایج هر قسمت استفاده می شود یعنی هرگاه داده اول درون latch بعد از اولین مرحله ذخیره شد می توان داده بعدی را وارد کرد.



$T_1 \quad T_2 \quad T_3 \quad T_4 \quad T_k$

تاخیر کل مسیر $= T_1 + T_2 + T_3 + \dots + T_k$

حداقل طول کلاک (دوره تناوب کلاک) $= Max(T_i) = T_p$



افزایش سرعت speed up $= \dfrac{nT}{kT_p + (n-1)T_p} = \dfrac{T}{T_p}$

$n \to \infty$

PAPCO

Subject:

Year. Month. Date. ( )

فرض کنید T_i ها برابر باشند نتیجه داریم:

$$T_p = T_1$$

$$T = kT_1$$

$$\rightarrow \text{Speed up} = \frac{kT_1}{T_1} = k$$

در نتیجه speed up به تعداد استیج‌هایی که اجرا می‌شود بستگی دارد و تنها به تعداد تقسیم‌بندی‌ها (stage) بستگی دارد و هرچه تعداد step ها بیشتر شود سرعت بیشتر می‌شود ولی این تقسیم‌بندی‌ها دارای محدودیت‌هایی است: ۱- مدار ترکیبی هر قدر قابلیت تقسیم شدن دارد. ۲- برای هر stage یک latch احتیاج داریم که هزینه و تأخیر اضافی را زیاد می‌کند.

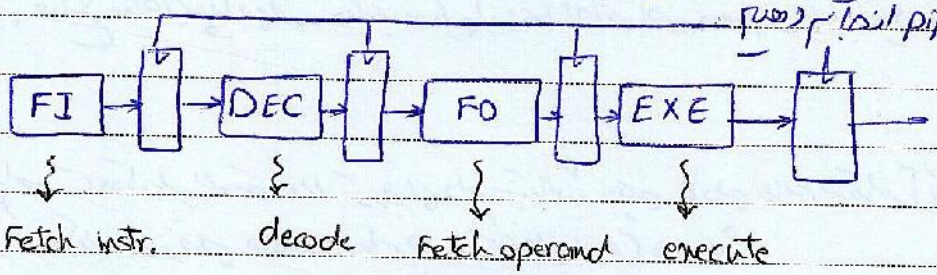
مثال: اگر در pipeline T_i ها با یکدیگر برابر نباشند، همانند شکل قبل باشد برای ۱۰۰ دستور speed up چقدر است؟

طول کد = ۱۰

$$\frac{23 \times 100}{40 + 99 \times 10} = \frac{2300}{1030} \simeq 2.2$$

این مقدار با تعداد مراحل برابر شد زیرا T_i ها برای هر دستور و n نیز به بینهایت بی‌نهایت نزدیک است.

حال می‌خواهیم ببینیم الگوریتم قبل چون به صورت sequential اجرا می‌شود را از طریق pipeline انجام دهیم.



میبینیم speed up آن برابر ۴ برای تعداد دستورات زیاد است.

Subject:
Year. Month. Date. ( )

نمودار زمان-مکان (spatial - Time diagram) وضعیت pipeline را در هر لحظه نشان می دهد.

	1	2	3	4	5	6	7	8
FI	T_1	T_2	T_3	T_4	T_5	/////	/////	T_6
DEC		T_1	T_2	T_3	T_4	T_5	/////	/////
FO			T_1	T_2	T_3	T_4	T_5	/////
EXE				T_1	T_2	T_3	T_4	T_5

واحد dec از زمان 2 شروع به کار کرده است ←

حباب ←

خانه های خالی نشان دهنده زمان هایی است که آن بخش بیکار است.

زمانی که یک خانه خالی در وسط موجود آید خانه های خالی در مراحل بعدی هم منتقل می شود و اصطلاحاً می گویند دچار حباب می شود.

علت به وجود آمدن حباب ها و مشکلات آن عبارتند از:

۱- دستورات انشعاب مثل jump: باعث می شود ندانیم دستور بعدی چیست و مشکل برای jump های شرطی است.

مثال: فرض کنید که یک دستور jump شرطی باشد باید به طور کامل اجرا شود تا بتواند دستور بعدی را وارد کند.

۲- resource conflict (تداخل منابع): در این جا stage 2 هستند که با ALU کار می کنند درنتیجه منابع تداخل پیدا می کنند و باید به یکی از آنها اختصاص دهیم و برای دیگری حباب بوجود می آید.

۳- ممکن است عملوند یک دستور به دستور قبلی وابسته باشد یعنی 2 دستور زمانی می تواند آرگومان خود را fetch کند که دستور قبلی محاسبه را به صورت کامل انجام داده باشد.

$R_1 \leftarrow R_2 + R_3$

$R_4 \leftarrow R_1 + 3$

$M[R_1] \leftarrow M[R_1] + 2$ ← data dependency مشکل وابستگی داده ای

Subject:
Year. Month. Date.

۳. برای برطرف کردن مشکل سوم از compiler کمک می‌گیریم در زمان ایجاد کد باینری آن تا جای امکان از تداخل دستورات جلوگیری می‌کند و دستوراتی که مفید است را زودتر اجرا می‌کنیم

۲. برای برطرف کردن مشکل دوم از چندین ALU استفاده می‌کنیم و افزایش منابع.

۱. گروه اول وابسته به کار است. برای برطرف کردن مشکل یکی از دو راه حل را ارائه می‌دهیم در صورتی که مسیر اشتباه است pipeline را پاک می‌کنیم و از ابتدا ارائه می‌دهیم. ضعف این روش این است که در صورت اشتباه بودن ما سه مرحله زمان تلف می‌شود که پیش‌بینی می‌کنیم به احتمال زیاد کدام اتفاق می‌افتد. برای اینکار از BTB استفاده می‌کنیم

Jump	100

BTB (branch Target Buffer)

محتمل‌ترین دستور را قرار می‌دهیم یک دستور و آدرس Jmp جمع را قرار می‌دهیم و در صورت فرا رسیدن دوباره همان دستور می‌توان از آدرس روبروی آن استفاده کنیم.

یکی دیگر از راه‌حل‌ها استفاده از (Branch prediction) BP است که متناسب با تعداد دفعات استفاده از یک دستور پیش‌بینی می‌کند که Jmp شرطی اجرا شود یا نه

راه‌حل سوم استفاده از compiler است و باید تا جای امکان از دستورات Jmp شرطی استفاده نکند. ممکن است Jump در کدها وجود داشته باشد و compiler آن را ایجاد کرده باشد.

Interrupt / IO :

نحوه connection دستگاه‌های IO به CPU دستورات خاصی نیاز دارد که In و Out نام دارد برای دسترسی به حافظه و خواندن آدرس از آن نیاز به Load و Store است:

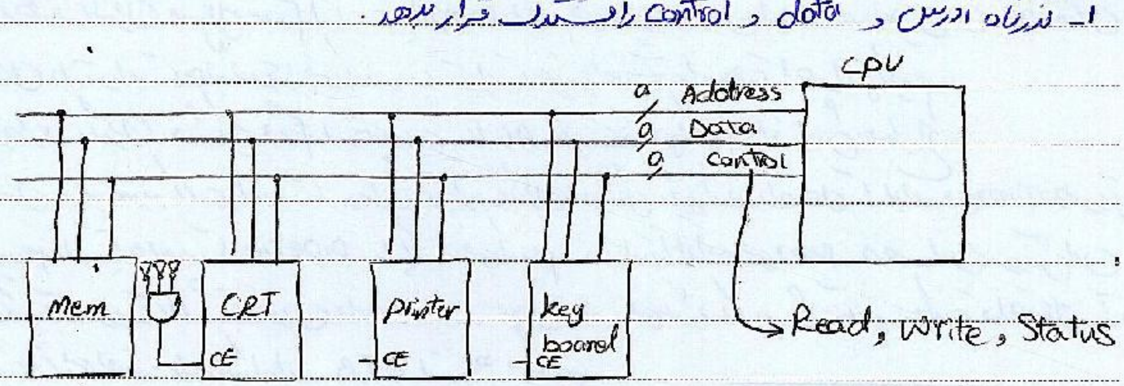
IO { IN آدرس
    Out آدرس

Mem { Load آدرس
    Store آدرس

سه روش برای اتصال IO ها به mem و CPU وجود دارد:

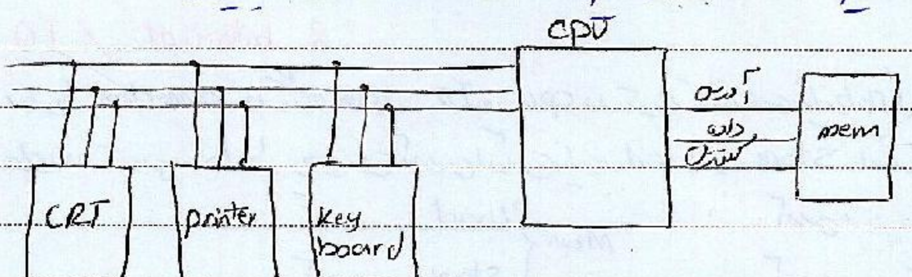
Subject:
Year. Month. Date. ( )

۱- گذرگاه آدرس و data و control را مشترک قرار بدهد.



هر کدام از این دستگاه ها دارای یک پایه chip enable هستند و مقداری که روی آدرس قرار دارد مشخص می کند داده و آدرس مربوط برای کدام یک از دستگاه هاست. در نتیجه آدرس ها بین ۴ دستگاه IO تقسیم می شوند و آدرس های اختصاص داده به بقیه دستگاه ها نمی توانند در mem به کار روند. به این نوع از اتصالات می گویند Memory-Mapped IO. در واقع اشاره به دستورات 0 و 1 و 2 از mem آدرس های CRT و printer و keyboard اختصاص داده شده است. یعنی دستورات مربوط به CRT و ... را هم می توان با Read و write از mem انجام داد. این روش از نظر سخت افزاری بهینه است ولی خانه هایی از حافظه را از دست داده ایم.

۲- گذرگاه های آدرس، داده و کنترل برای IO ها و mem جدا باشند.
در این روش تعداد سیم ها را دو برابر کرده ایم و مدار control را تغییر می دهد.



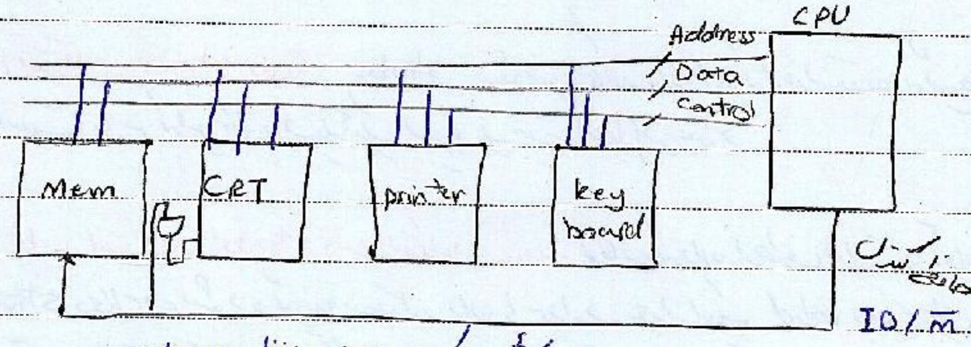
با وجود ساده تر بودن مدار در این روش ولی به علت سیم کشی زیاد و پایین آمدن توان قابل استفاده نیست.

PAPCO

دستگاه های IO و mem کلاک نمی‌پذیرند چون سرعت کامپیوتر (CPU) بسیار بیشتر از سرعت IO و mem است.

Subject:
Year. Month. Date. ( )

۳- گذرگاه آدرس و داده یکسان ولی گذرگاه کنترل جدا می‌شود. ← Isolated I/O



یک باس کنترل جدا برای حافظه وجود دارد.

از طریق سیگنال IO/M̄ مشخص می‌شود یا mem فعال می‌شود یا دستگاه های IO و یکی بودن آدرس، داده و control نشان دهنده یکی بودن mem و IO نیست و آدرس ها با یکدیگر conflict ندارد.

شیوه های تبادل اطلاعات:

۱- سنکرون: علاوه بر وجود پایه های enable هر دستگاه اجزای CPU به clock وصل می‌شود.

۲- آسنکرون: نیازی به clock ندارد و خروجی بعد از فعال شدن ورودی فعال می‌شود.

آسنکرون پرکاربردتر از سنکرون است زیرا یک سیم کشی clock کمتر دارد و در سنکرون تغییر موقع clock ممکن است انجام نشود.

از نظر دیگر شیوه انتقال داده را می‌توان به صورت زیر تقسیم بندی کرد:

۱) سری: اگر 8 بیت بخواهد منتقل شود در 8 زمان پشت سر هم منتقل می‌شوند.

۲) موازی: ~~[illegible]~~ به تعداد بیت ها سیم کشی می‌شود تا به طور همزمان همه اطلاعات را بتوانیم بفرستیم. در این روش سیم کشی زیاد است.

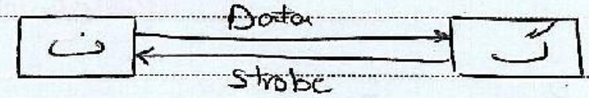
Subject:
Year. Month. Date. ( )

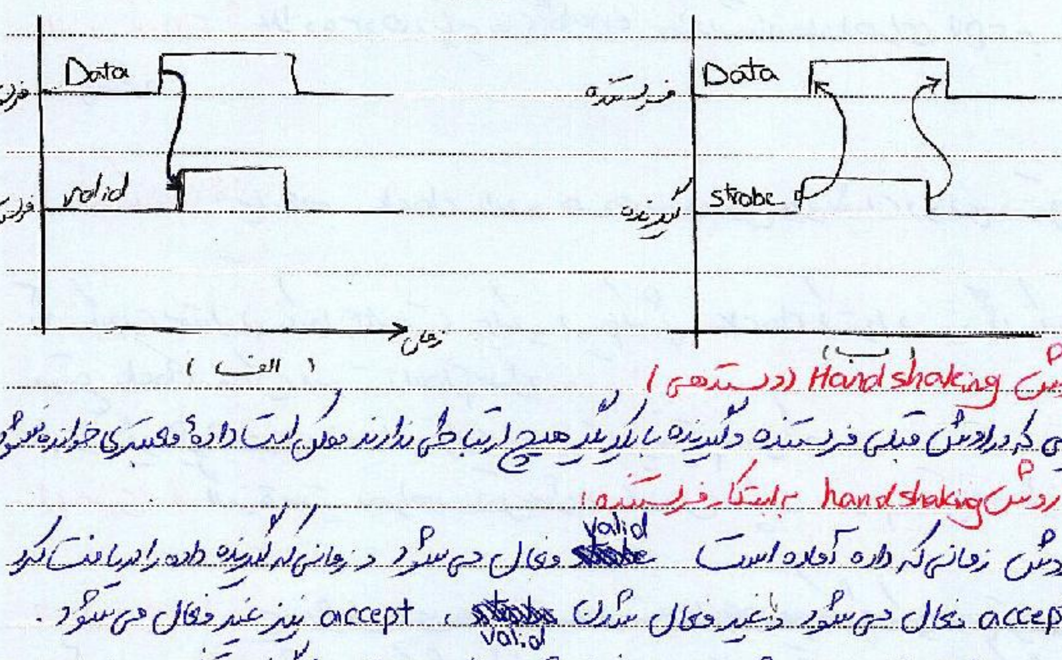
هشتمین نوع انتقال داده آسنکرون سری است.

الف) انتقال داده strobe: strobe مشخص می‌کند داده‌ای که از فرستنده به گیرنده می‌رود معتبر است یا نه که این کار بوسیله یک سیم یک بیتی انجام می‌شود.

الف ۱ - انتقال strobe دار به ابتکار فرستنده: زمانی که داده از طرف فرستنده آماده باشد strobe فعال می‌شود و گیرنده می‌تواند داده را بردارد. بعد از اینکه valid بودن برای گیرنده کافی است strobe غیرفعال می‌شود و داده فرستاده نمی‌شود. در این روش strobe بوسیله فرستنده تعیین می‌کند.

ب 2 - انتقال strobe دار به ابتکار گیرنده: هر زمان که گیرنده یک داده نیاز دارد strobe فعال می‌شود و داده قابل دسترسی است و بعد از استفاده از داده strobe غیرفعال می‌شود.





(الف) (ب)

ب ۱ روش Handshaking (دست‌دهی)

از آن جایی که در روش قبلی فرستنده و گیرنده با یکدیگر هیچ ارتباطی ندارند معلوم نیست داده معتبری خوانده می‌شود.

ب ۱) روش handshaking به ابتکار فرستنده:

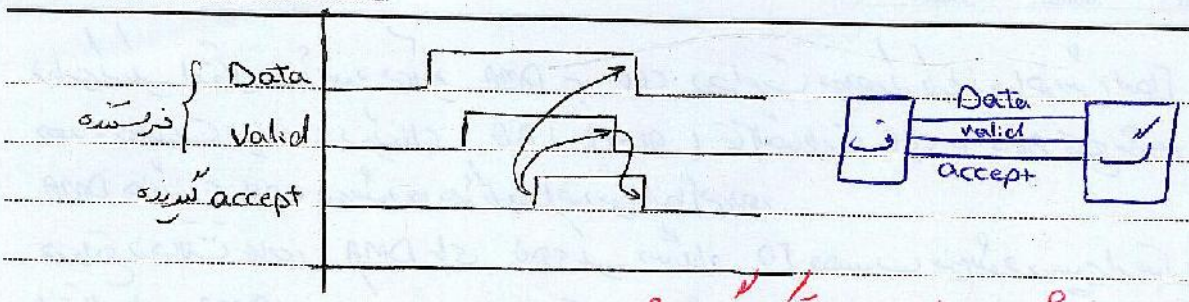
در این روش زمانی که داده آماده است valid فعال می‌شود. در زمانی که گیرنده داده را دریافت کرد پاسخ accept فعال می‌شود. با غیرفعال شدن valid، accept نیز غیرفعال می‌شود.

valid بوسیله فرستنده فعال شده و پس از فعال شدن accept توسط گیرنده متناسب با زمان مورد نیاز برای برداشتن داده valid را غیرفعال می‌کند و accept نیز غیرفعال می‌شود.

PAPCO

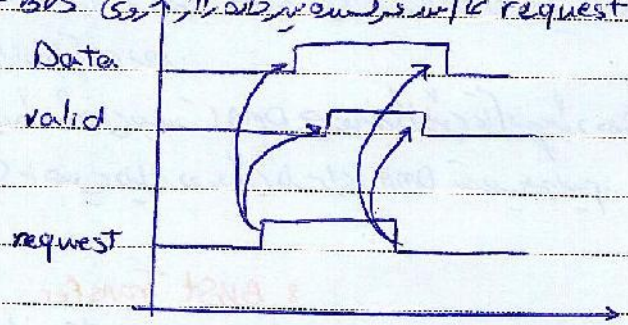
Subject:

Year. Month. Date. ( )



ب) 2 روش handshaking برای گیرنده:

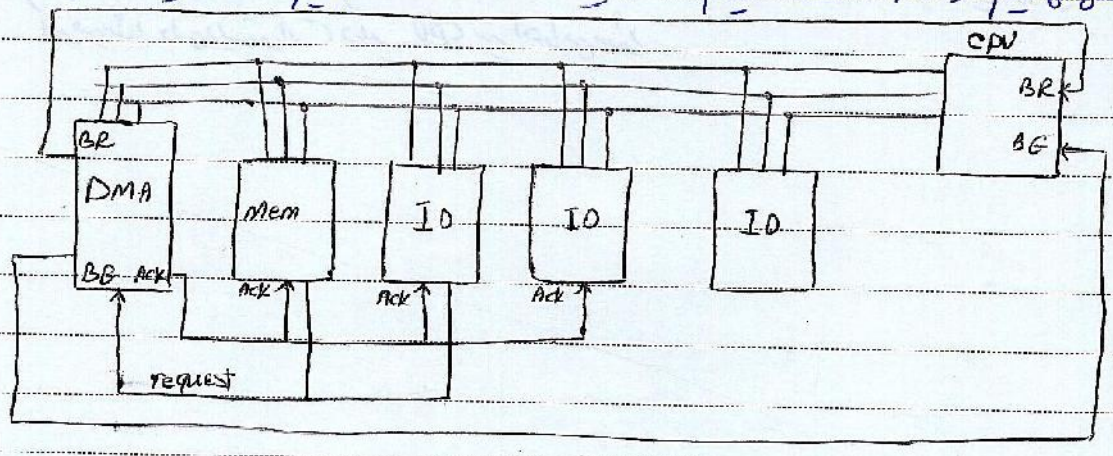
اگر از طرف گیرنده request آمد فرستنده داده را روی BUS قرار می‌دهد و پایه valid نیز فعال می‌شود. هر موقع سیگنال request تمام شد فرستنده نیز داده را از روی BUS برمی‌دارد



/

DMA (Direct Memory Access):

دسترسی به حافظه به صورت مستقیم است. هیچ گونه ارتباطی از mem به ID و برعکس به صورت مستقیم انجام نمی‌شود و داده باید از CPU بگذرد و دسترسی انتقال پیدا کند. از آن جایی که زمان بسیار زیادی هدر می‌رود و CPU استفاده خاصی از آن نمی‌شود در انتقال داده‌های انبوه (Burst) سعی می‌کنیم از CPU استفاده نکنیم و از یک DMA استفاده می‌کنیم که DMA نیز خود یک ID است



Subject:

Year. Month. Date. ( )

زمانی که یک انتقال داده آماده می‌باید DMA به CPU درخواست می‌دهد که کار آماده را خودش انجام دهد و در صورت دریافت سیگنال BG (grant) تمام اتصالات CPU به BUS قطع شده و DMA جایگزین CPU می‌شود و همه کارها را خودش انجام می‌دهد

در واقع در حالت عادی DMA برای CPU یک دستگاه IO محسوب می‌شود و پس از قبول انتقال داده DMA خود به یک زیر CPU تبدیل می‌شود علت تغییر دادن DMA این است که یک واسط افزاری است و تنها از طریق gate پایه‌های write , read دستگاه‌های IO را set می‌کند ولی CPU برای انتقال داده باید Instr داشته باشد و با اجرای دستورات این کار را انجام می‌دهد.

از آنجایی که مشخص نیست DMA چه زمانی کارش تمام می‌شود همه کارهای CPU تعطیل شده و این مسئله سازگار نیست به همین دلیل دو روش کلی برای DMA تعریف می‌کنیم:

I) 1. Burst Transfer

زمانی که کارها تمام شد BG را CPU می‌دهد در این حالت سرعت DMA بیشتر است ولی کارهای CPU تعطیل می‌شود

II) 2. Stealing cycle

در هر cycle ، DMA ، BG را به CPU می‌دهد کارهای خود را انجام دهد در این روش کار DMA کند می‌شود ولی دستورات دیگر CPU نیز اجرا می‌شود

Subject:

Year. Month. Date. ( )

8 Interrupt

از زمانی که یک درخواست داده می‌شود برای حل کردن اینکه آیا خروجی آماده است یا نه از دو روش زیر می‌توان استفاده کرد:

- سرکشی polling: مرتب دستور IO را صدا می‌زند که آیا خروجی آماده شده یا نه. با این روش آهسته زیاد است ولی نیاز به سخت افزاری ساده‌ای دارد.

- وقفه Interrupt: یک درخواست داده می‌شود و CPU به کارهای خود ادامه می‌دهد و زمانی که خروجی IO آماده شد پایه Interrupt فعال می‌شود. از دو روش زیر شخص می‌شود که Intr مربوط به کدام IO است.

Intr ها از نظر نحوه مشخص کردن نوع آدرس دهی به دو دسته زیر تقسیم می‌شوند:

I) Vector: عدد مشخص شدن پایه Interrupt در clock بعدی مقدار یا شماره Int فرستاده می‌شود.

II) Non_vector: شماره وقفه مشخص نیست و باید آدرس و شماره وقفه را از جایی از پیش تعیین شده در حافظه بخواند که این آدرس معمولاً صفر است.

Intrupt ها می‌توانند دو گونه زیر تقسیم شوند:

I) External: Interrupt هایی که از خارج از CPU توسط IO ها داده می‌شوند به کمک است برای کنترل دقیق پایه‌های Inti از نوع vector استفاده کنیم.

II) Internal: دون نیاز از Intr استفاده شده است و فرمت call به Intr این است که می‌تواند اجرای دستورات را متوقف کند و فوری اجرایی شود. که به دو نوع H/W و S/W تقسیم می‌شود.

HW: عیاتی که نمی‌خواهیم اجرا شود مانند تقسیم بر صفر که در این حالت در داخل ALU است.

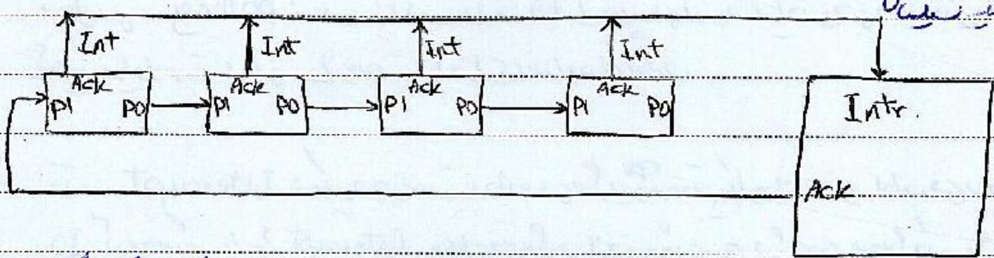
SW: مثلاً شماره intr 50 را اجرا کن که این حالت به صورت vector است. INT 50

Subject:
Year. Month. Date. ( )

برای اینکه مجبور نباشیم به ازای هر دستگاه IO یک پایه .intr در CPU قرار دهیم یک پایه قرار می‌دهیم و به گروه‌های یک .intr ها را با یک پایه وصل می‌کنیم. CPU با این نوع vectore باشد.

Daily chain

هر دستگاه‌ها .Intr خود را روی یک پایه .Intr از CPU قرار می‌دهند و هر وقت CPU موقعیت را برای اجرای .Intr مناسب دید یک سیگنال

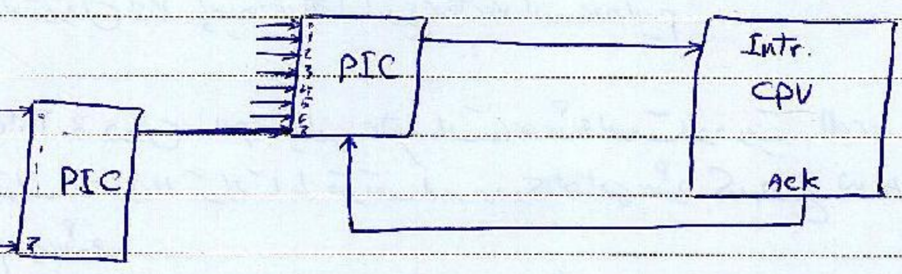


از طریق Ack به اولین دستگاه می‌فرستد. دستگاه‌ها براساس اولویت قرار دارند اولین دستگاه چک می‌کند که .Intr داده یا نه اگر .Intr داشت پیش آورده تا از آن استفاده کند. اگر درخواست نکرده به دستگاه بعدی می‌دهد و همین کار برای بقیه ادامه دارد. در این روش به تعویق افتادن می‌شود و ممکن است .Intr یکی را دیگری استفاده کند.

parrallel Intr.

در این روش همه .Intr های IO به دستگاهی به نام PIC متصل می‌شوند که وظیفه آن انتقال .Intr از IO به CPU در صورت فعال شدن Ack است. در صورتی که چند دستگاه با هم درخواست Intr بدهند اولویت‌بندی می‌کنیم.

اگر تعداد دستگاه‌هایی که به PIC وصل می‌شود از ۸ بیشتر باشد در این صورت یک PIC دیگر به این متصل می‌شود.



مثال: مقادیر ثبات های R_1 و R_2 و محتویات حافظه قبل و بعد اجرای یک دستورالعمل نشان داده شده است. دستورالعمل را مشخص کنید.

Before executing — after executing

	Before executing		After executing
R_1	F000	−2	EFFE
R_2	0034	+2	0036

mem (before):

Address	Content
32	F238
→ 34	046A
36	101B
	⋮
→ EFFE	3122
F000	1A23

mem (after):

Address	Content
32	F238
→ 34	3122
36	1018
	⋮
EFFE	3122
F000	1A23

چون کامپیوتر ۱۶ بیتی است با بید PC، ۲ تا اضافه بشود پس Auto dec / inc با دو تا کم یا زیاد عمل می کند

mov $[R_2]+$, $-[R_1]$

محتوای آدرس R_2 عوض بشود بعد محتوای R_2 دو تا اضافه بشود.

اول دو تا از محتوای R_1 کم کند بعد به R_2 اضافه شود.

مثال: با توجه به دستور زیر که R_1 را در مکانی از حافظه ذخیره می کند با توجه به مقادیر زیر آدرس موثر کدام است؟

Store R_1, Index, @R_2

Index=10

R=1000

۱۵	۱۰۰۰	۱۰۱۰	۱۱۶۰	۱۱۹۰	۲۰۱۰
۱۵	۲۰۰۰	۱۵۰	۳۰	۵۸۱	۱۷۰

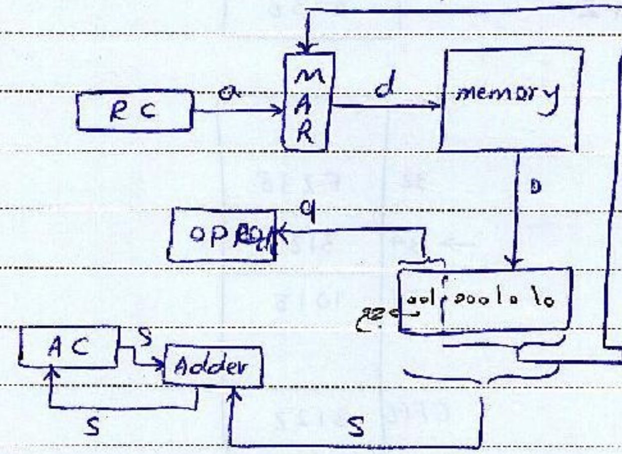
$[R_2] \rightarrow 2000+10=2010 \rightarrow$ | 170 |

آدرس مؤثر در R ذخیره می‌شود

store R_1, (index) R_2 اندیس

مثال: فرض کنید کامپیوتر عملیات اجرای یک دستورالعمل را به پایان رسانده. توسط سیگنال های علامت دار انتقالی که برای اجرای دستورالعمل های بعدی که عملیات جمع با آدرس دهی مستقیم نشان دهید.

Add (10)



operand تنها یک بیت Indirect بستگی دارد و هیچ ارتباطی با operator ندارد پس q و r می‌توانند همزمان انجام شوند

$a \rightarrow d \rightarrow D \rightarrow r \rightarrow q \rightarrow d \rightarrow D \rightarrow S$

مثال: یک ماشین دارای دستورات ۱۶ بیتی و آدرس حافظه ۶ بیتی می‌باشد بعضی از دستورات تک آدرسی و بعضی دو آدرسی است اگر n تا عمل 2 آدرسی داشته باشیم حداکثر تعداد دستورات عملی تک آدرسی چند تاست؟

IR 16

۴	۶	۶

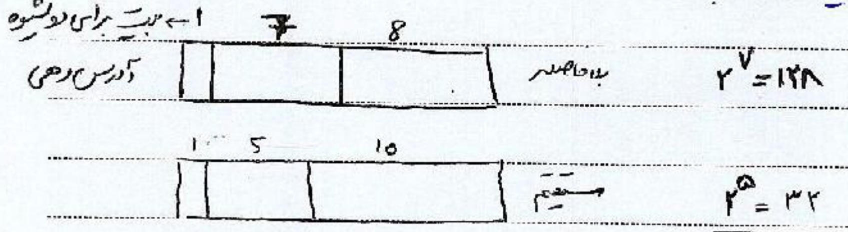
$(16-n) \times 2^6$

xxxx	۶

با 4 بیت opcode n تایی که در دو آدرسی استفاده شده است را نمی‌توان در تک آدرسی استفاده کنیم و آن بیت که در حالت قبلی آدرس یک محسوب می‌شد جزو opcode می‌شود.

Subject:
Year. Month. Date. ( )

مثال: در یک ماشین کامپیوتری از دو شیوه آدرس‌دهی مستقیم و بلافاصله استفاده می‌شود. طول ثبات AC ۸ بیت، طول ثبات IR ۱۶ بیت و طول ثبات AR ... بیت است. حداکثر تعداد دستورات ماشین چند بیت می‌تواند باشد؟

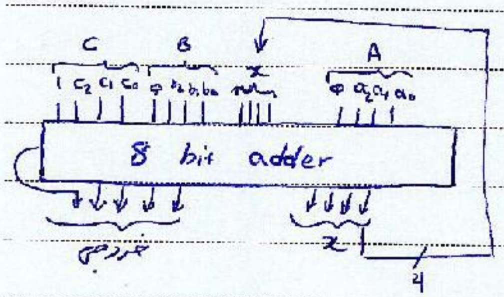


$2^7 = 128$

$2^5 = 32$

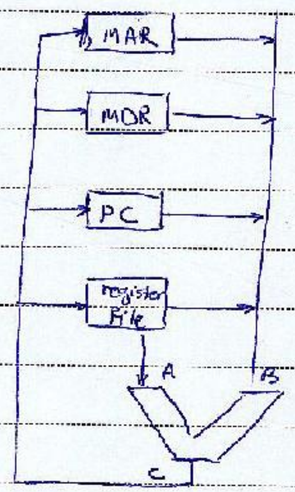
۱۶۰

مثال: این الگوریتم چه عملی را انجام می‌دهد؟



A + B + C + 9

مثال: برای دستور JR R_1 (lable، jump relative) چه مقادیری داخل A و B قرار می‌گیرد.



در آدرس‌دهی نسبی آدرس هر lable براساس PC مشخص می‌شود.

PAPCO