« معماری »

قلمرو معماری کامپیوتر

1. چگونگی طراحی یک کامپیوتر
2. تعیین نوع های مختلف دارد
3. نحوه دسترسی به حافظه
4. فضای آدرس دهی حافظه
5. لیست دستورات

Desired Design = { cooling, packaging, availability, performance, energy, cost, Integrated Circuit Design, implementation, Logic Design, organization, instruction set Design }

« سیر تکاملی کامپیوتر »

1. کامپیوترهای مکانیکی
2. ساخته شده از دامپ خلأ
3. به کارگیری ترانزیستور ← ↑ سرعت و ↓ فضا و حجم
4. مدارهای مجتمع ICها
5. VLSI

قانون Moore : هر ۱۸ ماه، تعداد ترانزیستورها در واحد سطح ۲ برابر می‌شود
مشاهده

انواع کامپیوترها

1. کامپیوتر شخصی (PC)
2. سرور Reliable
3. سیستم های کامپیوتری تجهیز شده فداگیرنده

warehouse computer

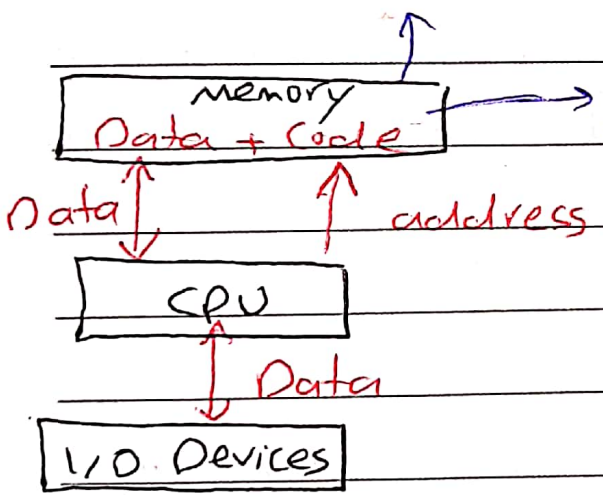
کامپیوتر دیجیتال

1. ثبات
2. گذرگاه
3. مدار ترکیبی
4. حافظه

انواع معماری کامپیوتر

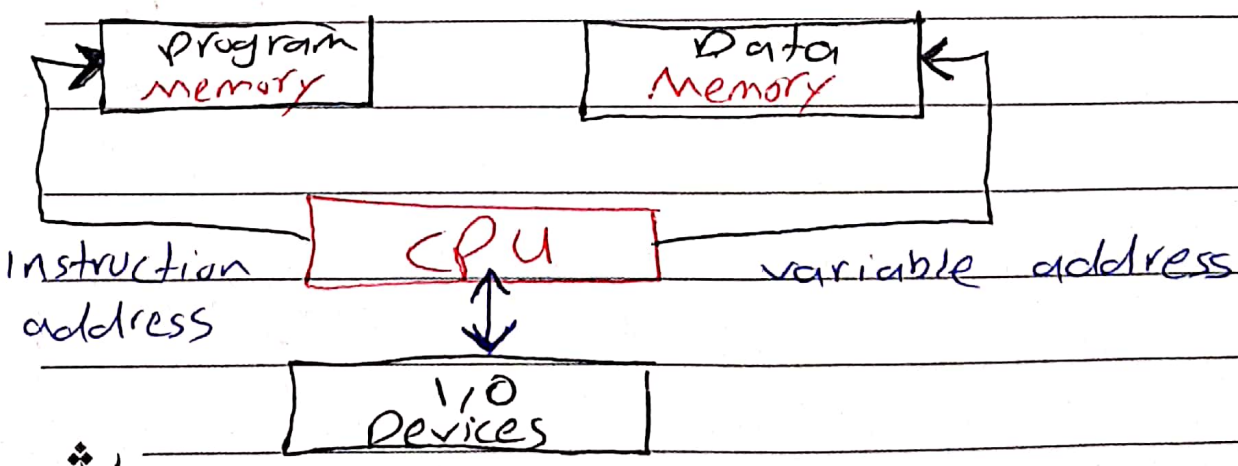
دستورات و داده ها

1. معماری Von Neumann



داخل یک حافظه فقط منفرد و قدرتی گیرد

2. معماری Harvard



Scanned by CamScanner

semantic Gap

1. Circuit (Logic)
2. Control — چه موقع چه کاری روی داده انجام شود
3. Instruction set Architecture — مجموعه دستورات
4. operating system
5. Assembly code
6. High level language
7. user Application

HW

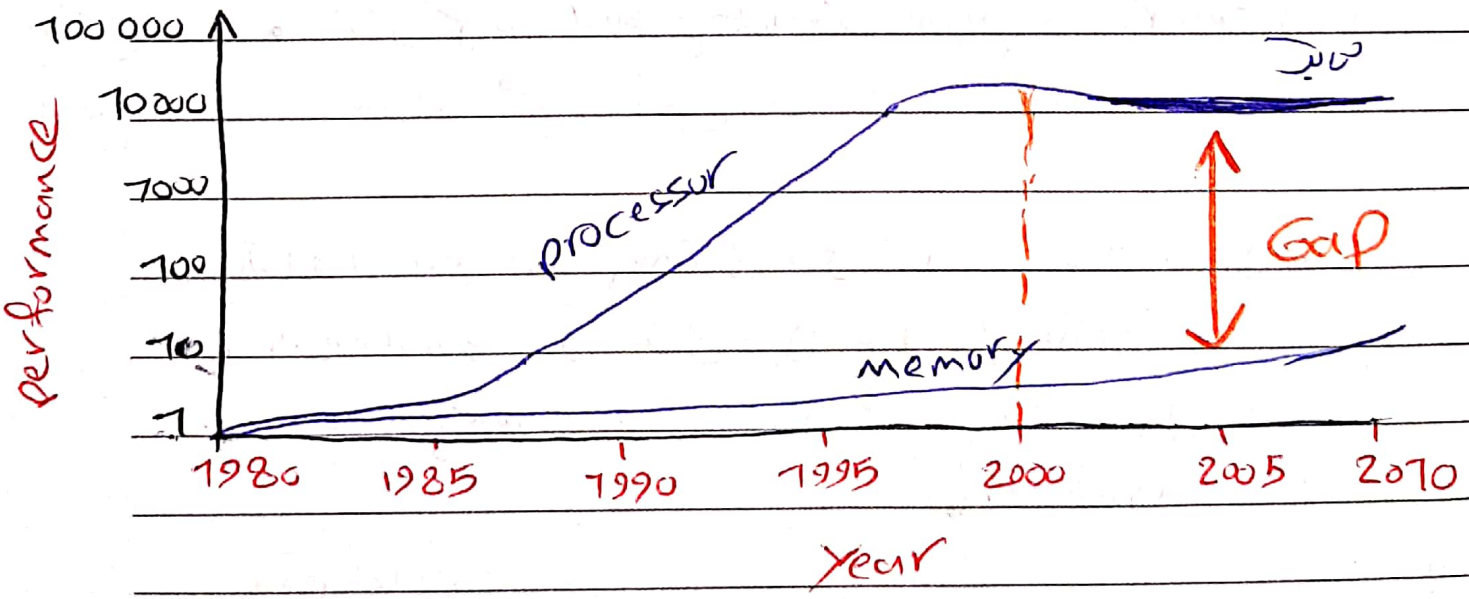
SW

دیدهای از Abstraction

در این دستگاه هر لایه از جزئیات لایه‌ی بالای خودش بی‌خبر است و فقط
دید از امکانات آن ارائه می‌کند

performance Gap

قانون مور برای صدرهای ترکیبی و پردازنده‌ها رخ داد اما برای حافظه به
دلیل ساختار خاص آن رخ نداد



← در این ۳۰ سال ← سرعت ۱۰ برابر بیشتر شده

پردازنده 6 10000 برابر بیشتر شده

نتیجه 6 گلوگاه در سیستم‌های کامپیوتری 6 حافظه

Data Representation systems

سیستم های نمایش اطلاعات در کامپیوتر

1. قرارداد برای تفسیر اطلاعات
2. اعداد در مبنای متفاوت
3. اعداد علامت‌دار و بدون علامت
4. اعداد صحیح و اعشاری
5. حروف و نشانه‌ها
6. کدهای انتقال
7. کدهای تشخیص و تصحیح خطا

قراردادها اهمیت فوق‌العاده دارند

اعداد در مبنای متفاوت

- مبنای r
- ارقام از صفر تا r-1
- ارزش هر رقم بجایگاه آن (n) بستگی دارد.

(قراردادها استثنا و تناقض نداشته باشد)

- ارزش کل عدد از مجموع حاصل ضرب هر رقم در r^n بدست می‌آید

پیدا کردن عدد نهایی

کیان: r^0 دهگان: r^1 و ...

- استفاده از روش تقسیم‌های متوالی برای تبدیل عدد دهدهی به مبنای r
- استفاده از ضرب‌های متوالی برای تبدیل بخش اعشاری
- مبناهای پرکاربرد: دسیمال، باینری، اکتال، هگزادسیمال

توانی از باینری

سیستم باینری

- مبنای ۲
- ارقام صفر و یک

$(1100101)_2 = ? \boxed{101}$

$2^0 \times 1 + 2^1 \times 0 + 2^2 \times 1 + 2^3 \times 0 + 2^4 \times 0 + 2^5 \times 1 + 2^6 \times 1 = 101$

(1 0 4 0 32 64)

Scanned by CamScanner

$(75) \xrightarrow{\text{در مبنای ۲}}$ تقسیم متوالی بر ۲

$(1001011)_2$

۷۵ | ۲
۳۷ | ۲
۱۸ | ۲
۹ | ۲
۴ | ۲
۲ | ۲
۱ | ۲
۰ | ۰
۱ | ۱

از راست به چپ

تا ۸ بیت حساب می‌کنیم

$75.82 = ؟ (1001011.11010001)_2$

$0.82 = ؟$ در مبنای 2

$0.82 \times 2 = 1.64 \longrightarrow 0.64 \times 2 = 1.28 \longrightarrow$

$0.28 \times 2 = 0.56 \longrightarrow 0.56 \times 2 = 1.12 \longrightarrow 0.12 \times 2$

0.24

...

* قسمت بعد از اعشار ۵ با ضرب‌های متوالی حساب می‌شوند نه تقسیم‌های متوالی *

کد بدون علامت

- لیستم کد بدون علامت همان کد باینری است صرفاً برای دامنه اعداد مثبت.
- کل بیت‌ها تعیین کننده اندازه کد هستند.
- دامنه اعداد قابل نمایش با n بیت ۵ از صفر تا $2^n - 1$

کد دو علامت

- در لیستم کد دو علامت یک بیت به کلمه چپ برای علامت اضافه می‌شود
- صفر برای علامت مثبت — یک بیت از کلمه چپ برای علامت
- یک برای علامت منفی

$+75 = \overset{+}{0}100\ 1011$

$-75 = \overset{-}{1}100\ 1011$

مکمل یک ← Not ← تمام ارقام

مکمل ۲ ← مکمل یک + ۱

دامنه برابر است با $[-2^{n-1}+1 \text{ و } 2^{n-1}-1]$

صفر مثبت و منفی $\longrightarrow$ $\underbrace{(-127 \text{ و } +127)}_{255 \text{ تا عدد}}$ $n=8$ دیگر

2^n-1 تا عدد

کارایی سیستم نمایش

با فرض n بیتی بودن کارایی یک سیستم برابر است با تعداد اعداد قابل نمایش در سیستم نسبت به تعداد کل حالات (2^n)

کارایی سیستم عدد عدد صحیح $= \frac{2^n-1}{2^n}$ $\quad \left(\frac{2^n-1}{2^n} < 1\right)$

* دهدهی و هیچ چیزی مبنی نیست *

مکمل ۲ و مکمل ۲-۱

فوق مهم

- مکمل ۲ یک عدد n رقمی برابر است با 2^n
- مکمل ۲-۱ یک عدد برابر است با مکمل ۲ آن منهای یک
- عدد مثبت بدون تغییر و با بیت علامت صفر نمایش داده می‌شود $(2^{n-1}-1)$
- عدد منفی به صورت مکمل ۲ و با بیت علامت یک نمایش داده می‌شود
- دامنه برای مبنای ۲ برابر است با $[-2^{n-1} \text{ و } 2^{n-1}-1]$

$\boxed{2^n}$ تا $\quad 2^6 = 64$ تا عدد توکی هست $\quad$ در بازه نیست $\quad$ قوس توی بازه هست

مکمل ۱۰ یک عدد مثل 3000 برابر $10^4 - 3000$

عملیات جمع و تفریق

- جمع به صورت بیت به بیت از سمت راست انجام می‌شود
- تفریق عدد B از عدد A به صورت جمع عدد A با مکمل ۲ عدد B انجام می‌شود

$A-B \xrightarrow{\text{می‌کنیم}} A+(B)^2 \xrightarrow{\text{مکمل ۲}} A+(2^n-B) = \cancel{2^n}+(A-B)$

دور ریخته می‌شود

$n+1$ بیت

صفر شود ← over flow نداریم

یک شود ← over flow داریم

XOR

• over flow از XOR دو بیت نقلی سمت چپ حاصل می‌شود

در سیستم بدون علامت ← اگر عدد خارج از دامنه قرار بگیرد ← over flow یا افتادن در محاسبات over flow هم داریم

۳۸

C_8 C_7

$53 + 23 =$

$$\begin{array}{r} 00110101 \\ + \; 00010111 \\ \hline 01001100 \end{array}$$

از راست بیت به بیت جمع می‌زنیم

$C_7 \oplus C_8 = 0 \rightarrow$ over flow نداریم

ex

$53 - 23 =$

53 + مکمل ۲ = 23

① -23 ؟

اول مکمل یک می‌گیریم

بعد با یک جمع می‌کنیم

$00010111 \xrightarrow{\text{مکمل یک}} 11101000 \xrightarrow{+1} 11101001$

مکمل ۲ = عدد ۲۳

C_8 C_7

② $$\begin{array}{r} 11101001 \\ + \; 00110101 \\ \hline 100011110 \end{array} = \boxed{+30}$$

دور ریز کرد بیت علامت = صفر

$C_7 \oplus C_8 = 0 \longrightarrow$ over flow نداریم

CA = 0 CV = 1

ex) 53 + 89 = ?

```
  0 1 1 1 1 1 1
  0 0 1 1 0 1 0 1
+ 0 1 0 1 1 0 0 1
  ---------------
  1 0 0 0 1 1 1 0
```

بیت علامت منفی شده ← 1 ← 1 0 0 0 1 1 1 0 ← کلمه ✗

$CV \oplus CA = 1 \rightarrow$

overflow داریم ← کدر مکمل دو است

باید کدر حاصل را بزرگتر در نظر بگیریم.

بیت بیشتر (9 بیتی)

اعداد اعشاری

fixed point.

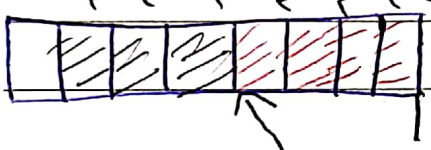
floating point.

IEEE 754

استاندارد

مثال ۲

علامت 2^{-4} 2^{-3} 2^{-2} 2^{-1} 2^{0} 2^{1} 2^{2} 2^{3}



مبنی

shift

مبنی

$[-2^{7}, 2^{7}) \rightarrow [-2^{+4}, 2^{4})$

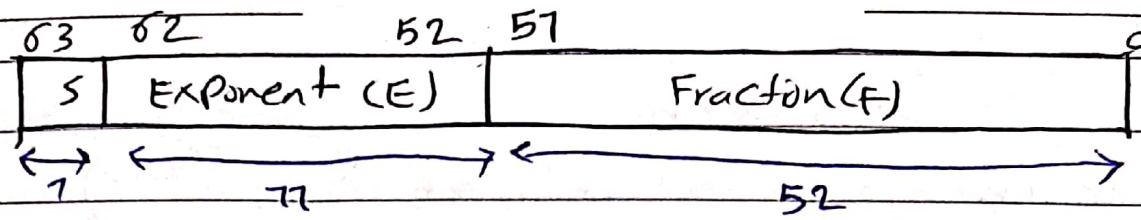
دامنه قبلی این بود

با تغییر مبنی $2^{\frac{0}{2}}$

فاصله بین دو کدر متوالی $\frac{1}{2^4}$ می‌شود ← دقت غالبی = $\boxed{\frac{1}{16}}$

هرچه مبنی را به کمت رالت ببریم ← دقت ↑ اما بازه کوچکتر می‌شود

~ ~ ~



64_bit Double_precision Floating_point number

ارزش کدر در سیستم Single precision :

$(-1)^{sign} \times 2^{(exponent-127)} \times (1+Fraction) =$ ارزش کدر

توان ۲ به ۱۵

۶۴ تا ۵

ارزش کدر ← Mantissa

normalize برای کردن : $M \xrightarrow{\text{تبدیل به}} M' \times 2^{e}$

به طوری که

$1 \le M' < 2$

۸ بیت ، توان ← می‌تواند + یا - باشد (علامت دار است)

ex : 01111100 01000000000000000000000

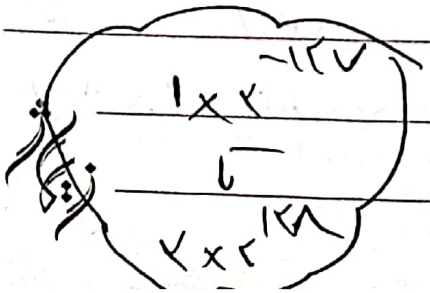
sign (1 bit) | exponent (8 bit) | fraction (23 bit)

$(-1)^{0} \times 2^{(124-127)} \times (1+\cancel{2^{-2}}) = 2^{-3} \times 1.25 =$

1.25

$\boxed{0.15625}$

کدر هیدکیاور



برای Single precision ← بازه‌ی اعداد

exp = ۰ → $2^{-127} \times 1$ کوچکترین عددی که می‌توانیم نمایش بدهیم

فاصله بین دو عدد متوالی به حسب exp متفاوت است

اگر exp = ۰ → $2^{-127} \times 2^{-23} = 2^{-150}$ فاصله خیلی کم

اگر exp = ۲۵۵ → $2^{+128} \times 2^{-23} = 2^{105}$ فاصله خیلی زیاد شد

★ دقت یعنی اختلاف دو عدد متوالی ★

تبدیل عدد اعشاری به single precision

ex

37.375 = ؟ 100101,011

حساب نمی‌کنیم

① 1.001011011×2^{5}

اول بایسری می‌کنیم

② عدد را نرمال می‌کنیم

چند تا بیت شیفت پیدا می‌کند ← توان = ۵

از اینجا را در fraction قرار می‌دهیم و باقی را صفر می‌گذاریم

③ بایاس توان $5 + 127 = 132 = (10000100)_2$

④ 01000010 00101100000000000000 ← جواب

exponent	Mantisa	value	حالت های خاص
0	0	exacto	مثل 2^{-168}
۲۵۵	0	infinity	قاعده‌ی بین 2 و 1
0	not 0	denormalised	نشرن ← ایاب
۲۵۵	not 0	Not a number (NAN)	

-127 و -128 در بازه‌ی [۱۲۸]

باشد ← اگر نبود denormalised می‌شد

مثل $\frac{255}{128 - 128}$ 0

ex1

```
#include "stdafx.h"
int i = 0x40A00000;
// 0100_0000_1010_0000_0000_0000_0000_0000
// exp = 2
// matisa = 1.25
float *f = (float *) &i;
float f2 = (float) i;
printf("%d", i);        → 1084227584
printf("\n%f", *f);     → 5.000000
printf("\n%f", f2);     → 1084227584.000000
```

* عملیات floating point کندتر از عملیات عادی fixed point انجام می‌شود
ولی چون تعداد عملیات کمتر از fixed است به یک نفر افزار جدا
برای این می‌گذاریم. (یک کمک پردازنده برای عملیات اعشاری)

ضرب اعداد اعشاری

$$\times \begin{matrix} (-1)^{s} \times 2^{e-127} \times (1+f) \\ (-1)^{s'} \times 2^{e'-127} \times (1+f') \end{matrix} = e$$

$$(-1)^{s+s'} \times 2^{(e+e')-127-127} \times \underbrace{(1+f)}_{1 \le f < 2} \times \underbrace{(1+f')}_{1 \le f' < 2}$$

e جواب ضرب.

$$= \underbrace{(-1)^{s \oplus s'}}_{\text{XOR}} \times 2^{e''-127} \times 1 \le f'' < 4$$

① علامت بدلت میاد

② بین ۱ تا ۴

④ نرمالایزم کنیم

③ نرمالایز

جمع اعداد اعشاری

$$+\ (-1)^{s} \times 2^{e-127} \times (1+f) \qquad = ?$$
$$(-1)^{s'} \times 2^{e'-127} \times (1+f')$$

برای جمع دو عدد ← ابتدا باید توان هایشان یکی باشند

پس اول باید توان ها را مقایسه کنیم

چون توی سیستم Bias آلت ←

حقیقی توان ها را یکسان کنیم ؟ یکی از توان ها بزرگ است و یکی کوچک

کدام را مبنا قرار دهیم برای یکسان سازی ؟ توان بزرگتر چرا؟

دوتا عدد را در نظر بگیر

یکی خیلی بزرگ و یکی خیلی کوچک ← اگر این دوتا را جمع نزنیم ← می‌توانیم

از عدد کوچک صرف نظر کنیم. (نمی‌ریم گرد کردیم)

بزرگ را مبنا قرار می‌دهیم

به همه توان بزرگتر یکسان می‌کنیم

mantis را به سمت راست به اندازه‌ی اختلاف دو توان، تغییر می‌دهیم

shift می‌دهیم

مزیت: این سیستم ؟ رنج خیلی بزرگی را می‌توانیم در یک فضای محدود نمایش دهیم

عیب: سرعت ؟ عملیات ها زمان بر و پیچیده است ← راه حل ؟ سخت افزار جداگانه برای Floating Point

از ضرب

یک مثال ؟

$A = 0\ \underbrace{07777700}_{\text{۸ بیت}}\ \underbrace{070}_{\text{۲۳ بیت}}$ ——— مانتیس

$B = 1\ \underbrace{00000001}_{\text{۸ بیت}}\ \underbrace{0101 0}_{\text{۲۳ بیت}}$ ———

X

Scanned by CamScanner

کد BCD

- برای نمایش هر رقم دهدهی ۴ بیت نیاز است.
- هر رقم جداگانه تبدیل می‌شود
- تبدیل اولیه عدد شامل گروه‌های ۴ بیتی برای ارقام مختلف است

* در ورودی خروجی، ما می‌خواهیم دهدهی نشان دهیم

75 = 0111 0101
7 5

* برای جمع دو عدد در BCD ۴ بیت ۴ بیت جمع می‌کنیم و با عدد ۹ که ۱۰۰۱ است مقایسه می‌کنیم اگر کمتر ۹ بود خروجی را می‌گذاریم اگر بیشتر ۹ بود ۱۰ تا از آن کم می‌کنیم و $\boxed{1}$ بیت نقلی به عدد بعدی می‌دهیم

0111 0101
+ 0100 0010
1011 0111 → 0001 0111 = 17 (جواب)

7 — 0001

کدهای نمایشی

- کد ASCII استاندارد که برای هر کاراکتر ۷ بیت در نظر می‌گیرد
- کد توسعه یافته ASCII ← ۶ بایتی (۸ بیتی)
- سیستم UNICODE «سمبل»

از ۹ بزرگتر ↓ ۱۰ تا از آن برمی‌داریم

۲ بایتی

lookup table

lookup پیدا کردن معادل در یک دیکشنری مثلاً

	معادل باینری
a	0111010
b	⋮
⋮	⋮
z	⋮
!	⋮
?	⋮

به جای ۲۵۶ تا کاراکتر ↓

در UNICODE ۳۶۵ تا کاراکتر می‌توانیم نمایش دهیم

برای تبدیل یک رشته که حروف کوچک دارد به حروف بزرگ مثلاً abc شود ABC

برای تبدیل a به A ↓

'a' - 'a' + 'A' = A

g → G

a → A

فاصله‌های آن

برای تبدیل g کوچک به G

'g' - 'a' + 'A' = 'G'

Scanned by CamScanner

پیدا کردن شماره‌ی کاراکتر ← '9' - 'a' ← شماره‌ی 9 را در اختیار می‌دهد (صفحه ۹)

کاراکتر بعد ← 'G' - 'a' + 'A' ← 9 تا بعد از A میشه G

از کجا بفهمیم کاراکتر کوچک بوده؟

کاراکتر = ch

در JS البته

```
if('a' <= ch <= 'z')
```
کوچک

```
if('A' <= ch <= 'Z')
```
بزرگ

'3' = '0' + 3 ← کاراکتر

'5' = '0' + 5

★ سوال: حروفی دو رشته‌ی '13' و '25' را جمع کنیم؟

```
  '13'
+ '25'
------
  '8'
```

'3' = '0' + 3

+ '5' = '0' + 5

'8' = ~~'0'~~ + '0' + 8

اگر کاراکتر '8' را نخواهیم ← '0' - باید کنیم

+ '0' + 8

session 6

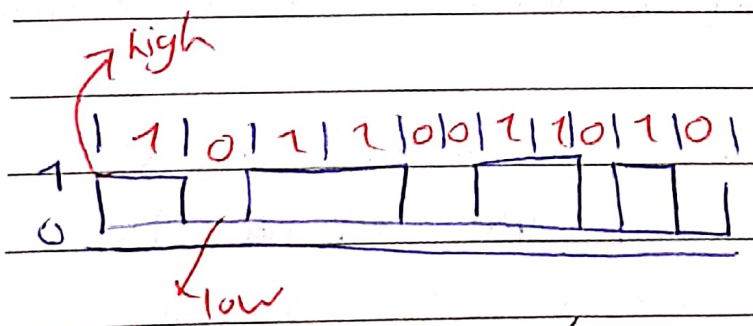
«کدهای انتقال»

دلایل انتقال داده: افت سیگنال در هنگام پالس

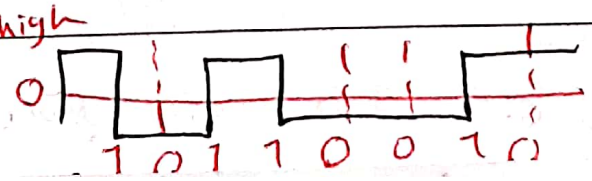
برای انتقال data از یک کامپیوتر مبدأ به مقصد

انواع کد:

① NRZ



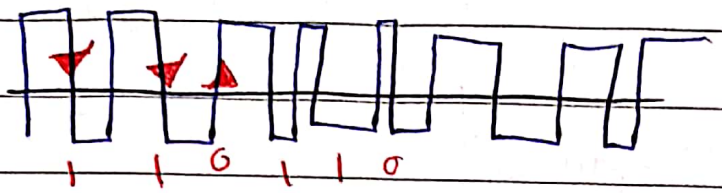
② NRZI صفحه پولاریته را تا بیت نمی‌دارد اما 1 عوض می‌کند



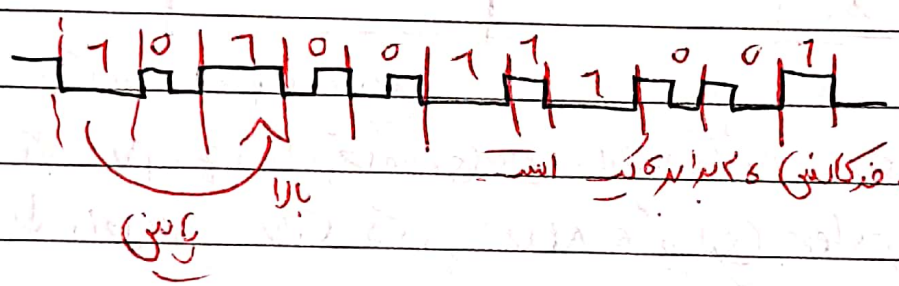
Scanned by CamScanner

③ PM (phase Modulation)

★ به ازای هر بیت، یک تغییر فاز ← یک را با لبه‌ی پایین رونده نشان بده و صفر را با لبه‌ی بالارونده



④ FM (frequency mod) به هر صفری یک لبه بده و هر یک نسبت به یک قبلی، پولاریته را عوض کن



به ازای صفر ← فرکانس، ۲ برابر یک است

کدهای ED ، EC

Error Detection → Error correction

دلیل التقاره، خرابی، نویز

انواع:

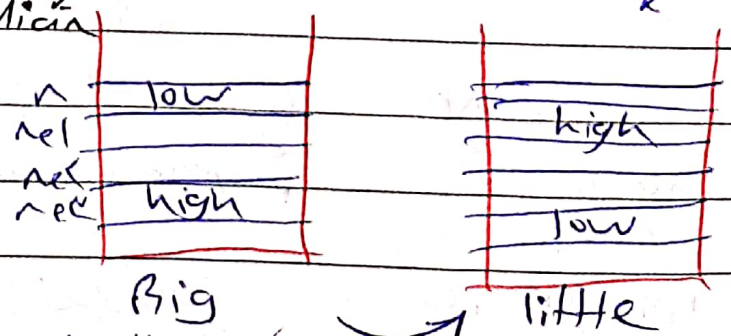
① بیت توازن ② CRC ③ کد همینگ

انواع پردازنده، خیلی دارد در حافظه

x86

Big Endian: بیت کم ارزش در آدرس بیشتر

Little Endian: بیت کم ارزش در آدرس کمتر



Endianness: Big → little

تفاوت ساختار معماری در انتقال داده

Scanned by CamScanner

Register Transfer language : RTL

زبان انتقال ثبات ها

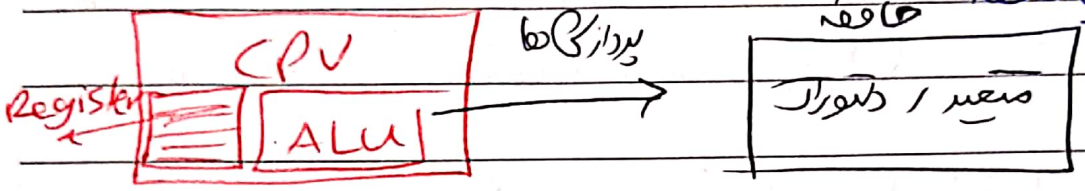
با تداخل دستور، باشد می شود ①

ثبات : تعدادی فلیپ فلاپ مرتبط به هم

یک الگو حافظه برای ذخیره ی یک بیت دیتا

② ذخیره ی بیت های کد دارد

③ سریع تر از حافظه



حافظه کند و CPU سریع

اگر CPU همیشه به حافظه ی جانبی وابسته باشد ← سرعتش را از دست می دهد

↓ داخل خود CPU کنار ALU یک سری register می گذاریم

تقریباً سرعت برابر سرعت CPU

چرا کل حافظه را با register پیاده نمی کنیم ؟ گران است

③ همه ی با مدارهای منطقی و محاسباتی

هم سرعت

انواع ثبات : ① General purpose : R_1 و R_2

در هر عملیاتی می توانند شرکت کنند با شمارنده نشان می دهیم

$R_1 + R_2 \rightarrow R_3$ کنند

② special purpose : PC و MAR

program counter — کارهای خاص — memory address register

special purpose : کمتر باشد

* مهم * شمارنده بداند که تعداد ثبات

هر چقدر general purpose بیشتر باشد راحت تر برنامه می نویسیم
(↓ محدودیت ها)
((بلوک دیاگرام ثبات))

R1	7 6 5 4 3 2 1
Register R	Showing individual bits

15 ... 0	15 ... 8	7 ... 0
R2	PC(H)	PC(L)

Numbering of bits — high — Divided into 2 parts — low

فایل ۸ کامپیوتر ما ۶۴ بیتی است اما می توانیم روشی، نرم افزار ۳۲ بیتی هم داشته باشیم

۳۲ بیت کم انرژی، یک register جدا ← هر موقع می توانیم با ۱۶ کار کنیم

micro operation (ریز عملیات)

* یک عملکرد پایه که بر روی داده درون ثبات صورت می گیرد
  - بارگذاری (load) • افزایش • شیفت • پاک کردن • جمع
* یک ریز عملیات یک یا چند کار را در یک پالس ساعت انجام می دهد

((تعریف کامپیوتر بر اساس ثبات ها))

- تعداد ثبات ها، نحوه ارتباط آن ها
- مجموعه ریزعملیات های ممکن بر روی ثبات ها
- منطق کنترل توالی ریزعملیات ها ← نرم افزاری

شیفت — افزایش

زبان RTL

• زبان نمادین برای تشریح کار ثبات ها

کارهایی که روی ثبات ها انجام می شود:

تبدیل ← مثل جمع و not و and و or

انتقال ← مقدار R1 را بریز توی R2

• از حروف بزرگ و کاراکتر برای نمایش ثبات ها استفاده می کنیم.

R1 ، R2 ، PC ، IR ، MAR ، ...

انواع ریز عملیات

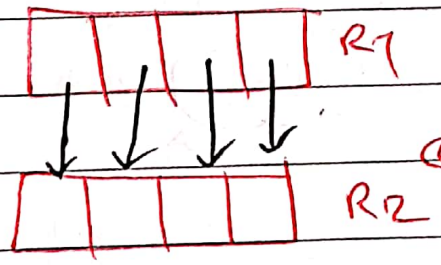
① انتقال اطلاعات ② شیفت ③ منطقی ④ ریاضی

انتقال: انتقال، بارگذاری یک کپی از اطلاعات ثبات مبدا به ثبات مقصد

با علامت ← نشان می دهیم

R2 ← R1 یعنی R1 را کپی کن و بریز توی R2

بیت به بیت



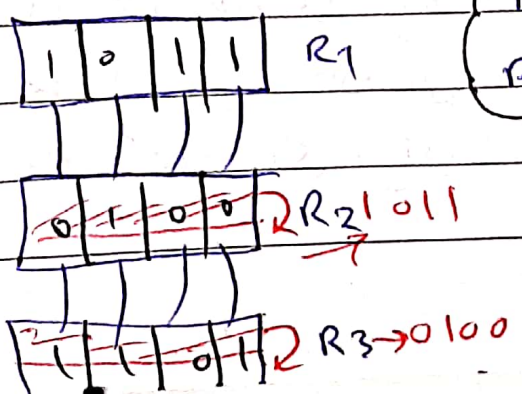
* نکته *

«در انتقال، مبدأ و مقصد باید هم اندازه باشند»

R2 ← R1
R3 ← R2

پس از این عملیات، مقدار R3 ← 0100

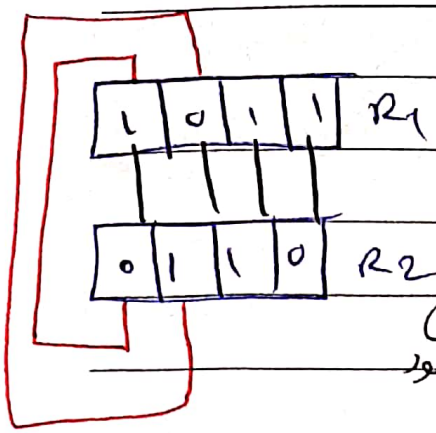
بدون تغییر مبدأ و مقصد دارای کوتاهی می باشد



* در سمت افزاره اول و دوم نداریم

هماهنگی همزمان رخ می دهد

چون register ها ۵ بخشی هستند ←



$R2 \leftarrow R1$

$R1 \leftarrow R2$

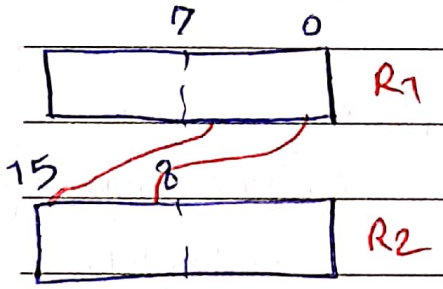
جای دو مقدار به هم عوض می‌شود

انتقال جزئی Partial Transfer

• انجام یک زیرعملیات بر روی بخشی از ثبات (انتقال بخشی از register)

• مثالی با استفاده از عملگر ()

$R2(15_8) \leftarrow R1(7_0)$



قاعده ۵ ابعاد بهتر حیث و رالت (بخشی که کپی می‌کنیم) باید با هم برابر باشند.

مثلاً ۵ ① $R2(0) \leftarrow R1(0)$

بدین

② $R1(9_8) \leftarrow R1(3_2)$

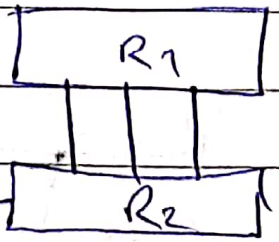
انتقال شرطی

• انجام انتقال در صورت تحقق یک شرط مشخص

• مثالی با استفاده از عملگر :

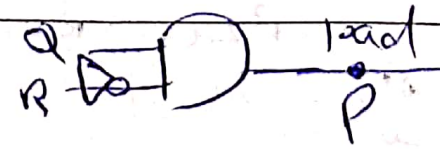
$P\,B\,R2 \leftarrow R1$

در شرط P، R1 را بدین توی R2



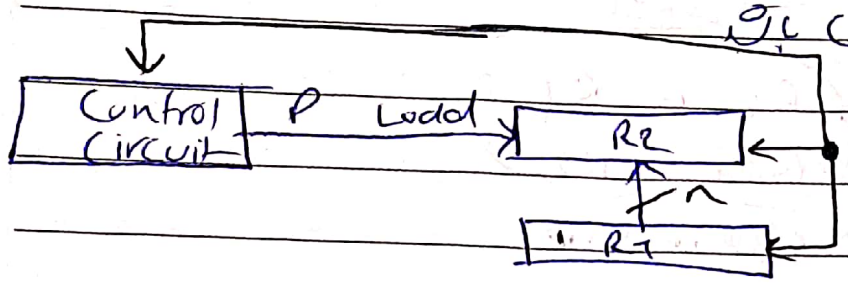
زیرگاه

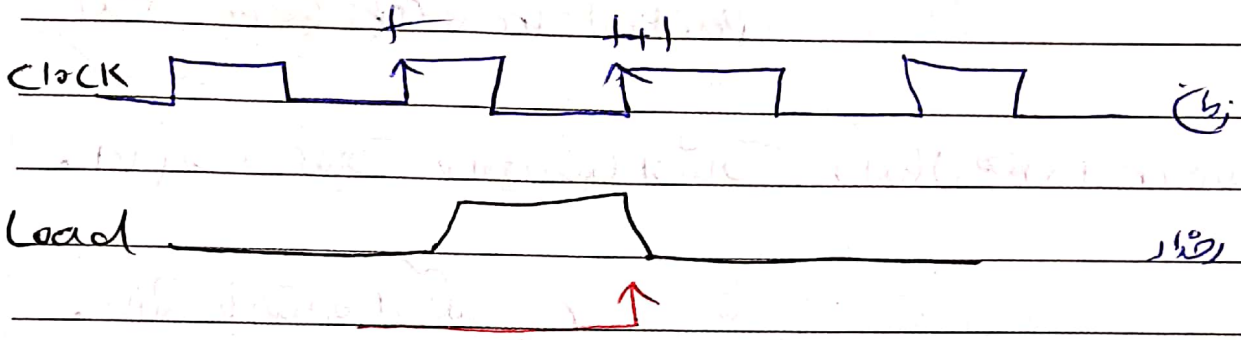
مثلاً $P = Q.R$



Scanned by CamScanner

انتقال کلی: شرط می‌تواند حاصل یک عملیات ترکیبی (بیت Carry) یا یک الارم زمانی مشخص (T2) باشد





$P \cdot T2 : R2 \leftarrow R1$

اگر P و وقتی T2 بود، R1 را برابر در R2

$R1 \leftarrow R1$ درست

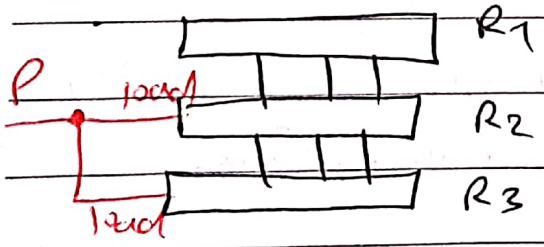
((عملیات همزمان))

• انجام چند انتقال به صورت همزمان

• نمایش با یک کلمه (6)

$P : R2 \leftarrow R1,\ R3 \leftarrow R2$

★ نکته: مقصد عملیات همزمان، حتماً باید متفاوت باشد



if هر موقع P رخ داد

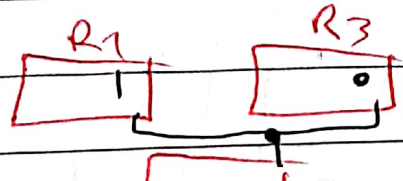
همزمان، R1 به R2 و R2 به R3 منتقل می‌شود

یعنی

$R2 \leftarrow R1$

$R2 \leftarrow R3$

غلط است



اگر مقدار یک داده باشیم ← انتقال کوتاه تغذیه و زمین

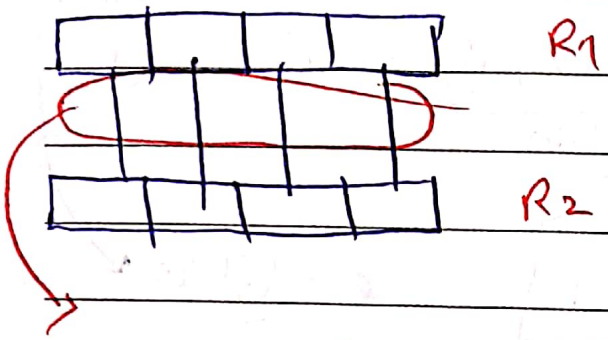
روشن شدن مدار زیگزاگ

Session 7

انتقال بر روی گذرگاه (BUS)

Bus ها تعدادی سیم با مبدأ و مقصد مشابه از یک ثبات که بیت های مختلف آن ثبات را انتقال می دهند

وقتی می گیم شبت افزاری داریم که R1 را می تواند به R2 انتقال دهد



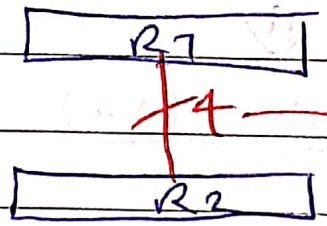
R2 ← R1

یعنی ع تا لیم ۱ به ازای هر بیت ۵ بیت لیم

مبدا هر ع تا = R1

مقصد هر ع تا = R2

ع تا لیم → برای ع تا لیم ← یک BUS می گیم



یک روش دیگر برای نمایش

ع تا لیم

گذرگاه مشترک :

① برای انتقال بین ثبات ها نیاز به تعداد کابل توتی سیم داریم

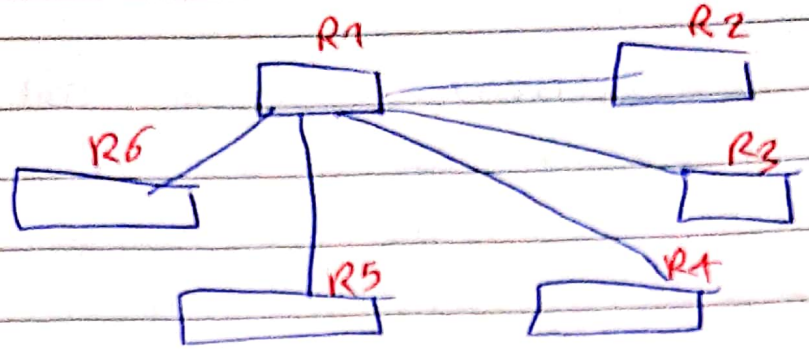
② به جای این کار از گذرگاه استفاده می کنیم

③ برای تعیین مبدأ و مقصد نیاز به منطق کنترلی است

④ انتقال بر روی گذرگاه با مالتی پلکسر

مثلاً ۴ تا reg داریم ← وقتی می گیم R منظوره یعنی باید بتوانیم هر کدام از reg ها را مبدأ و یا مقصد قرار بدهیم

Scanned by CamScanner



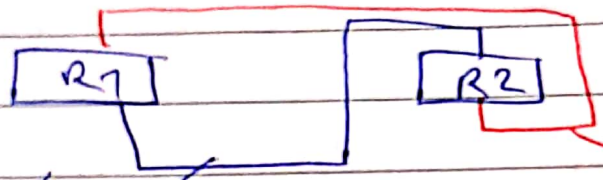
یک گراف کامل ← باید درست کنیم بین reg ها

$n(n-1)$ تا لینک

می‌خاییم ← خیلی زیاد می‌شه ← راه‌حلی نیست

هزینه‌ی زیاد

چرا این‌طوره؟



برای اینکه R7 را ببندیم توی R2

برای اینکه R2 را ببندیم توی R7

⇓

$\frac{n(n-1)}{2}$ نداریم ← می‌شه $n(n-1)$

تقسیم بر ۲ نداریم

برای ۶ تا رجیستر ←

$8 \times 5 \times 6$ ← تعداد لینک‌های مورد نیاز

تعداد لیت‌ها

ساختار IC ما ۲ بعدی است ← این لینک‌کشی‌ها باید روی این ساختار پیاده‌سازی شوند ← گراف باید مسطح باشد ← یعنی هیچ دو یالی برخورد نداشته باشند

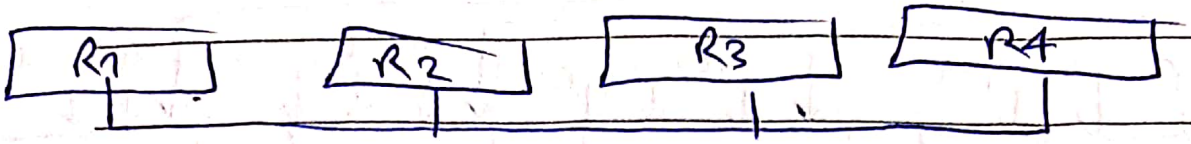
نمی‌گیم یک گراف کامل می‌خواهیم بدهیم یک گراف مسطح

گراف کامل تا درجه‌ی ۴ مسطح است

مشکل بعدی این نوع لینک‌کشی ← باید IC را چند بعدی کنیم

← می‌توانیم بین هر ۲ نقطه دلخواه reg بزنیم

به جای این کار ما یک گذرگاه مشترک می‌سازیم



توی این اختیار ← در هر لحظه ← فقط یک صدا می‌تونه داده منتقل کند

چند تا یکی روی یک لیم 1 می‌نویسه و یکی (یک reg دیگر) روی همان لیم 0 می‌نویسه ←

انتقال 1 به صفر ← انتقال کوتاه ← اکیپ به مدار نقطه ضعف نسبت به گذرگاه کامل ما

سرعت کمی بیشتر افت پیدا می‌کند

اما مقدار می‌تواند چند تا باشد ← این BUS ، یک برنامه‌ی Broadcast نیست

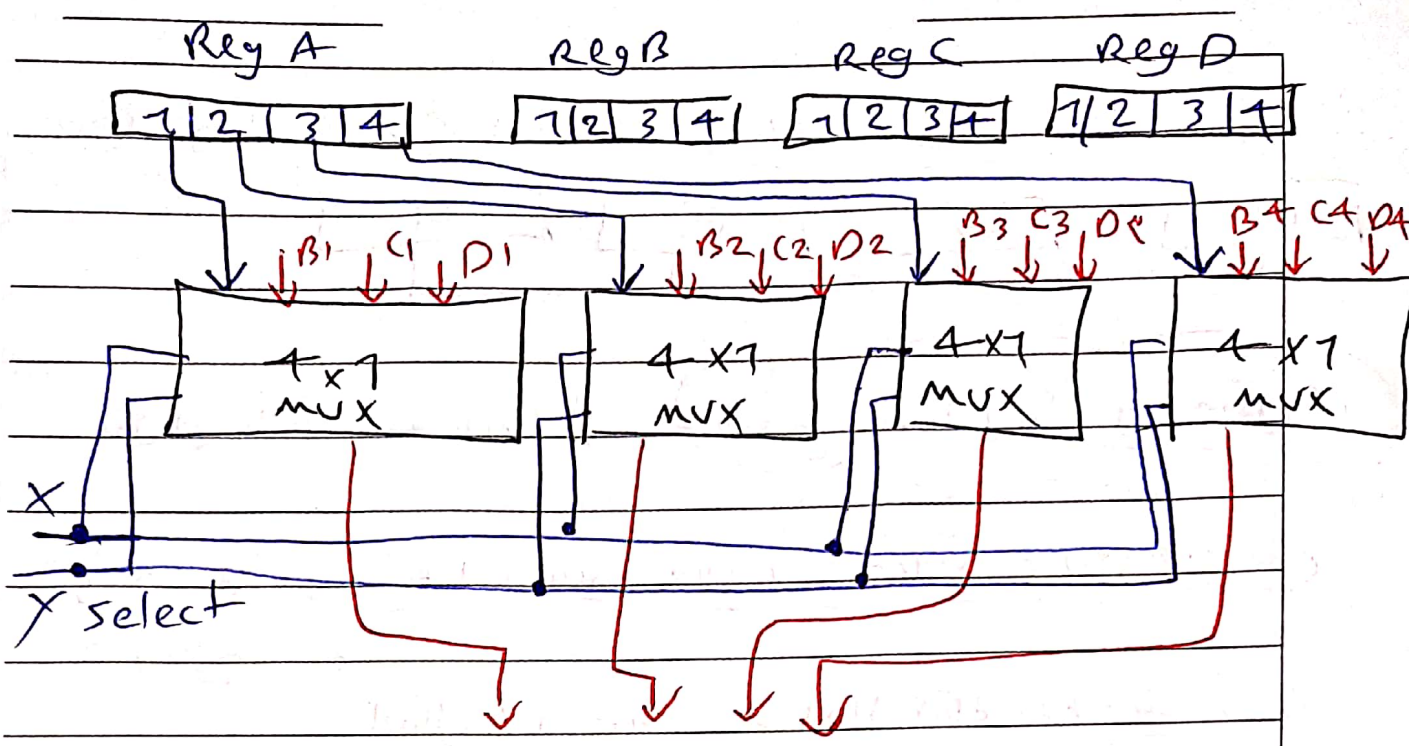
یعنی به همه می‌فرستد و کسی نمی‌تواند جلویش را بگیرد

اولین روش برای کنترل صدا ، انتقال بر روی گذرگاه به کمک مالتی پلکسر

فرض کنید 4 تا reg ، 4 تا بیت داریم می‌خواهیم با یک باس (BUS) 4 بیتی بهم متصل کنیم.

باید به تعداد بیت‌ها ، مالتی پلکسر بگذاریم

کدوم مالتی پلکسر از کجا می‌آید ؟ از تعداد reg ها



4 line bus

$A_1 A_2 A_3 A_4$

$\boxed{x=0 \text{ و } y=0}$

در MUX اول ← A1

~ ~ دوم ← A2

~ ~ سوم ← A3

~ ~ چهارم ← A4

مثل این است که reg ← $\boxed{A}$ را به bus وصل کردیم

تعداد طبقات مالتی پلکسرها ← از تعداد بیت های reg ها معلوم می‌شود

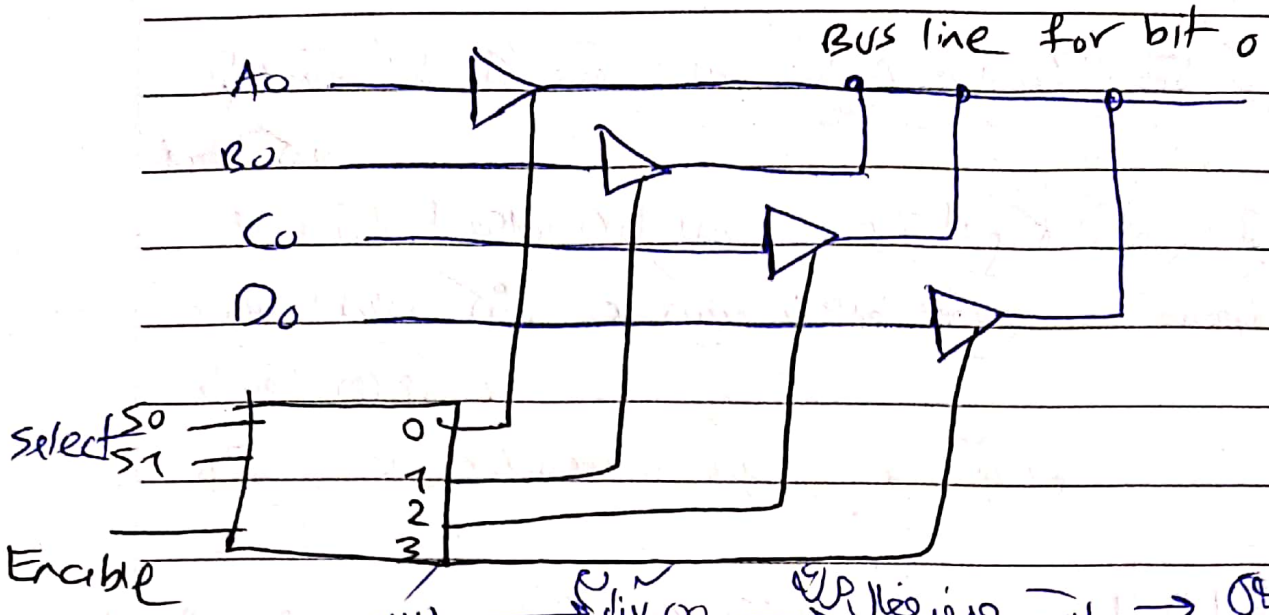
تعداد مالتی پلکسر ← از تعداد reg ها

باید حواسمون باشه طوری تقسیم کنیم که در هر لحظه یکی وصل شود

با باز ۳ حالته هم می‌تونیم این مدار را پیاده‌سازی کنیم؟

با باز ۳ حالته ← یک خط انتخاب دارد که اگر بالا باشد ← وارد بی‌خطری وصل است اگر هم پایین کنیم ← قطع می‌شود

صفر



دیکدر → 00 بگذاریم → بیت صفر فعال می‌شه → باقی خانه‌ها اولی فعال می‌شه و بقیه قطع می‌شن

01 → روی فعال می‌شه

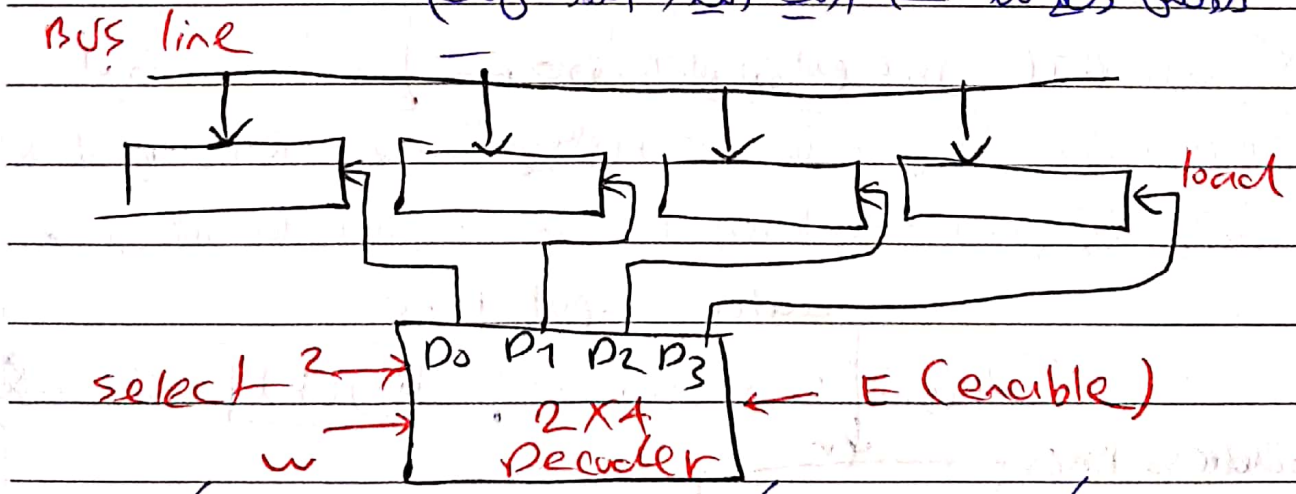
11 → چهارتا فعال میشه

10 → روی فعال می‌شه

Enable برای چیه؟ بتونیم هر کی را از bus قطع کنیم

برای مهم

باید مشخص کنیم که کدام reg قرار است دیتایی که روی bus هست را دوباره ذخیره کند ← از یک دیکدر استفاده می‌کنیم



به الاکی اینه ورودی های دیکدر را چه بدهیم مثلاً load یکی از reg های بالا فعال می‌شود ← می‌تواند اطلاعاتی روی bus هست را بخواند.

Scanned by CamScanner

CPU ← دیتاهایی را از کجا می‌آورد؟ یا از حافظه یا از ورودی خروجی

برای مقصد ← اگر فقط یک دیکدر بگذاریم ← در هر لحظه فقط یک reg می‌تواند load کند

اما می‌تواند broadcast کند ← می‌توانیم کاری کنیم یکی از reg ها دیتا را دریافت کند ← دیکدر را حذف می‌کنیم ← مسیرها و load E را مقدار دهی می‌کنیم

می‌توانیم به جای decoder یک reg بگذاریم

load R2, R3 → مدل reg را اگر این طوری ← | 0 | 1 | 0 | 1 | (E پس)

همزمان فعال می‌شوند و دیتا را می‌خوانند

نمایش به صورت جبری:

R2 ← | BUS , BUS | ← R1

چون می‌دانیم یک گذرگاه این ولتاژ را می‌توانیم بنویسیم

نمایش به صورت فنی

R2 ← R1

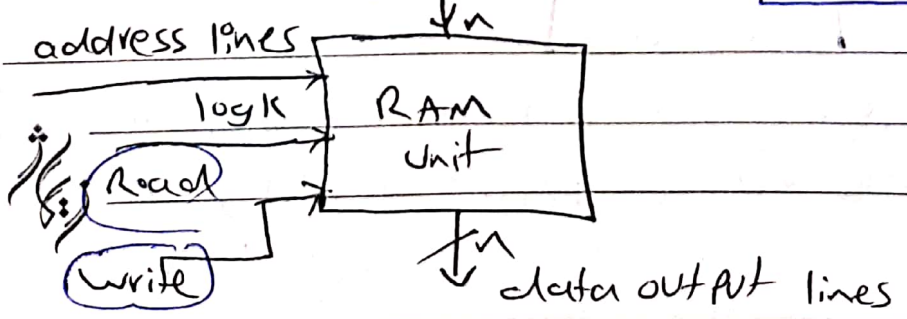
انتقال حافظه ← حافظه را باید بتوانیم با زبان RTL توصیف کنیم

★ از دید RTL ← حافظه مجموعه‌ای از ثبات‌ها به همراه یک مدار کنترلی است

★ این ثبات‌ها با آدرس‌های متوالی از صفر تا K-1 از هم متمایز می‌شوند

data input lines

★★★ نیاز به $\log(K)$ خط آدرس در مدار کنترلی است



این بار داده در حافظه از صفر تا 1023 از هم متمایز می‌شوند

← نیاز به $\boxed{\log 1024}$ خط آدرس داریم ← نیاز به 10 تا خط آدرس داریم

Read = 1 یک reg الان ← خواندن از حافظه

write = 0 R1 ← M[MAR] → memory address register

memory

بخوان ← بریز توی رجیستر $\boxed{R1}$

نوشتن در حافظه ←

M[MAR] ← R2

Read = 0

write = 1

جدا می‌گیم M[5] و جدا می‌گیم M[MAR]

چرا یک reg اشاره کردیم و

RTL یک زبان توصیف سطح افزار است

ما با کی برنامه‌نویسی می‌کنیم ← اگر بنویسیم M[5] یعنی همیشه فقط همین را می‌توانیم بخوانیم

→ ثابت است

ما می‌خواهیم خانه‌های مختلف را در زمان‌های مختلف بخوانیم ← به جای اینکه ثابت بنویسیم باید یک $\boxed{reg}$ بنویسیم که بتواند مقدار آدرس را تغییر دهد

MAR ← یک reg ← خاص منظورک

شیفت به راست B

منطقی A از بعد چپ یک صفر وارد کن و یک بیت از راست خارج کن

shift (به چپ و راست)

ریاضی B شیفت به راست ← مثلاً تقسیم بر ۲

کدر B ← به چپ ← ×۲

زیگزاگ

اگر عدد مکمل ۲ باشد ← شیفت صحیح کاربردی ندارد (معنی نمی‌دهد)

shift ← ریاضی ← بیت علامت را توی خودش کپی کن و از بیت علامت به بعد با شیفت برو.

10001101 مکمل ۲ الف → $\boxed{\text{شیفت ریاضی}}$

$\boxed{1}1000110$ → مکمل می‌گیریم و به عدد دودویی تبدیل می‌کنیم

بیت علامت

$$\begin{array}{r} 00777007 \\ + \quad 7 \\ \hline 00777070 \end{array} \rightarrow \boxed{-58}$$

مکمل

حالا ← ارتباط 58 با 775 چیه؟

$\frac{775}{2} = 58$ $\quad \left\lfloor -\frac{775}{2} \right\rfloor = \boxed{-58}$

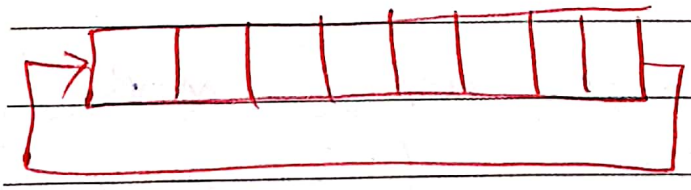
← شیفت ریاضی به راست ← واقعاً تقسیم بر ۲

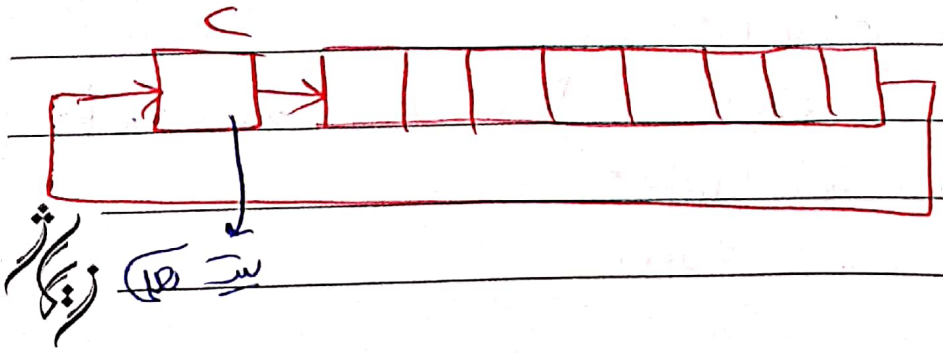
← ← چپ ← ضرب در ۲

$\boxed{\text{Rotate}}$ ← عدد را می‌چرخانیم

① بدون بیت نقلی

② به همراه بیت نقلی





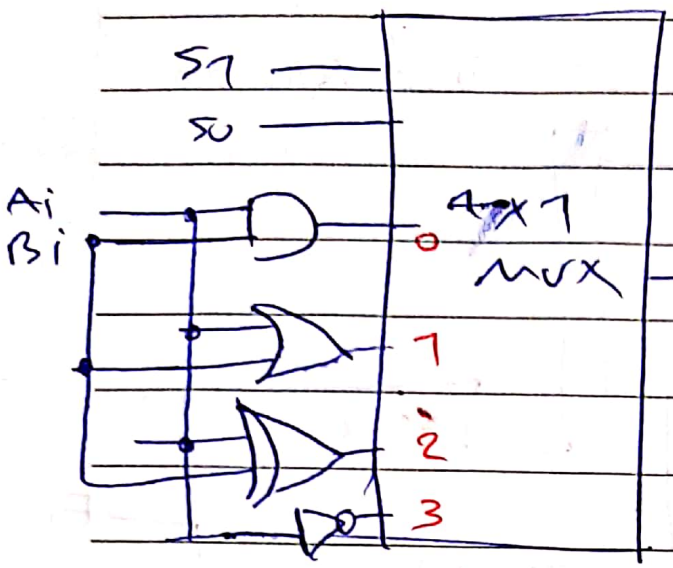
Scanned by CamScanner

Session 8

ساختار یک طبقه از مدار منطقی

این عملیات منطقی

And, or, xor, not



S_1	S_0	E_i
0	0	$A_i \wedge B_i$
0	1	$A_i \vee B_i$
1	0	$A_i \oplus B_i$
1	1	$\overline{A_i}$

RTL ← مربوط به آنالوگ

$\overline{S_1}\,\overline{S_0}$: $E_i \leftarrow A_i \wedge B_i$

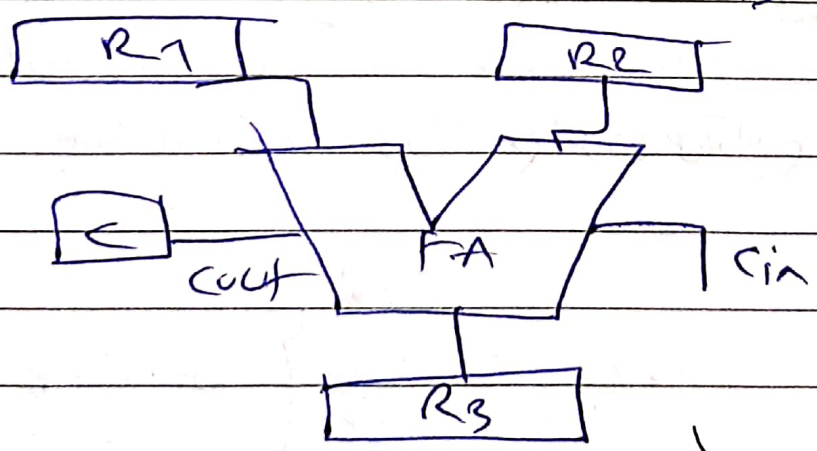
به شرط

این عملیات انجام می‌شود

$\overline{S_1}\,S_0$ و $S_1\,\overline{S_0}$ و $S_1\,S_0$:

$\boxed{\text{مثال}}$

$R_3 \leftarrow R_1 + R_2$



تمرین $R_3 \leftarrow R_2 - R_1$ در پست

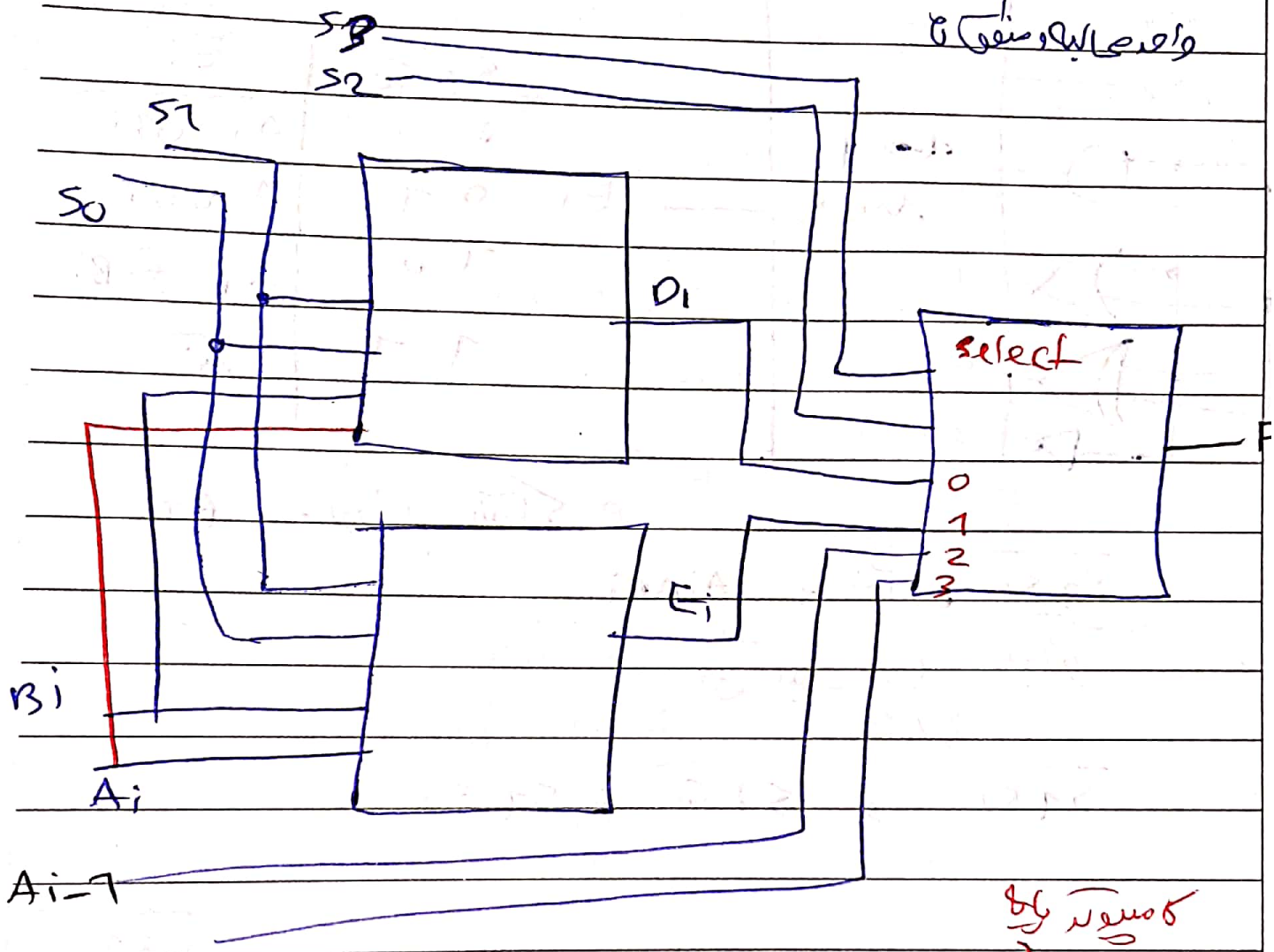
چون کمک افزار تفریق

نداریم

زیرنویس

$R3 \leftarrow R2 + \overline{R2} + 1$

واحد محاسبه و منطق



کامپیوتر پایه

تنها با استفاده از نیمجمع و And و Not و جمع می‌توان هر محاسبه‌ای را انجام داد.

محاسبه‌های دیگر را به صورت نرم‌افزاری می‌توانیم بنویسیم

کامپیوتر کامل:
- ALU
- محاسبه و منطق و پردازنده
- حافظه
- ورودی خروجی

Scanned by CamScanner

کار با I/O ← مثال حافظه Memory mapped ✓ اکثر
دستورات اختصاصی I/O Mapped

یک حافظه داریم ← n تا خانه دارد ← هر کدام از این خانه‌ها باید آدرس منحصر داشته باشد
n تا خانه باید بسازیم

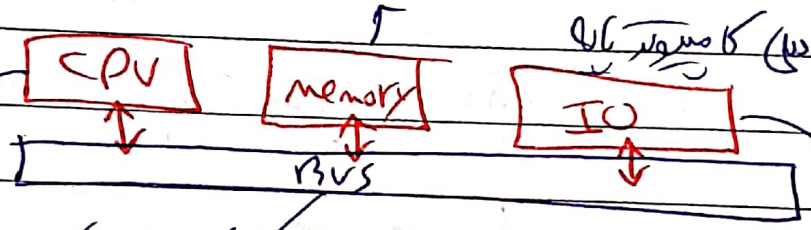
$2^n = n \rightarrow \boxed{n = \log n}$

حداکثر می‌خواهیم تا بتوانیم با log n بیت، n تا آدرس مکان بسازیم

session 9

حافظه‌ها مجموعه‌ی داده‌ها و دستورات را به پردازنده می‌دهد و نتایج را تحویل می‌گیرد

اجزای اصلی کامپیوتر یا



پردازنده‌ی مخفف اصلی سیستم

گذرگاه مشترک

کار با صفحه کلید و ...

یک بخش از CPU، کنترل گذرگاه را بر عهده دارد
CPU به حافظه می‌گوید که بخش می‌تواند data بردارد و بگذارد

مدیریت BUS

مدار دنباله‌ای

ساختار CPU

Data path (ALU, Registers)	Control path (Control unit)

CPU

شامل reg ها و مدارهای ترکیبی است که وظیفه‌ی پردازش زیادی داده‌ها

ALU → واحد محاسبه و منطق

FAI محاسبه مثل logic ← یک And و Xor و ...

Scanned by CamScanner

چیزی که مدار دیجیتال را تبدیل به کامپیوتر می‌کند، واحد کنترل است

Control path ← یعنی پایه‌ی نرم‌افزار و پردازش در سخت‌افزار قرار می‌گیرد

برنامه:
مجموعه‌ای از دستورات که در خانه‌های متوالی حافظه قرار می‌گیرند

دستور: یک مجموعه از بیت‌ها است که تعیین می‌کند چه کاری انجام شود

یک قرارداد ← تولید مهار لیستم تعریف می‌شود

واحد کنترل با دریافت دستور به واحدهای سخت‌افزاری فرمان می‌دهد

به مجموعه‌ی دستورات یک کامپیوتر: ISA می‌گوییم

واحد کنترل ← دستورات را از کجا می‌گیرد؟ از حافظه

شکل کلی دستورالعمل

fetch | decode | execute

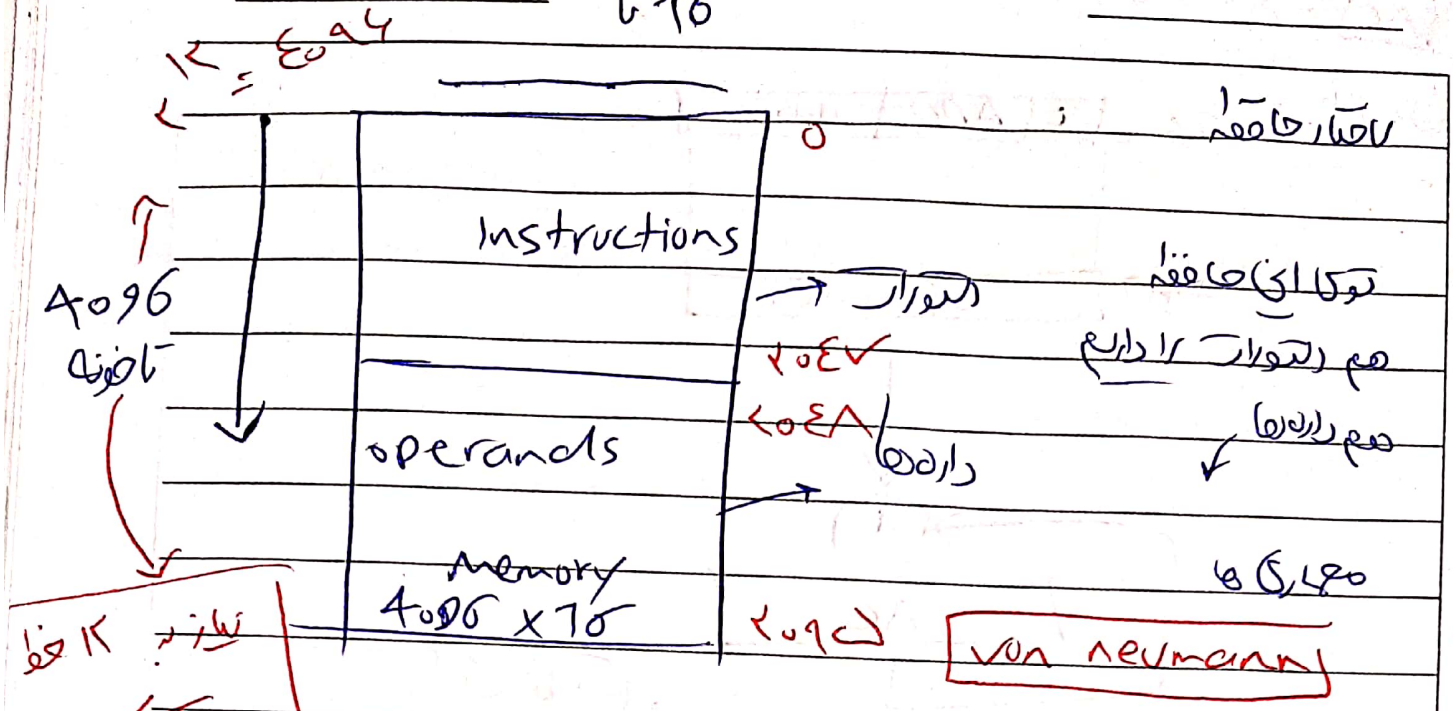
مرحله‌ی 1 ← دستور از حافظه به CPU منتقل شود ← fetch (یعنی واکشی)

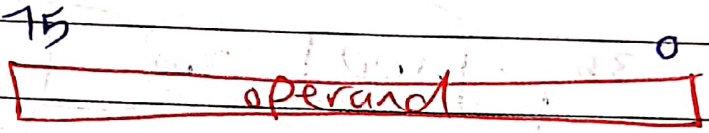
2 ← باز کردن قرارداد
تا بفهمیم توی هر کدام از بایت‌ها چی هست

مرحله‌ی 3 ← اجرای دستورات

4 ← میریم برای دستور بعدی (خانه‌ی بعدی حافظه)

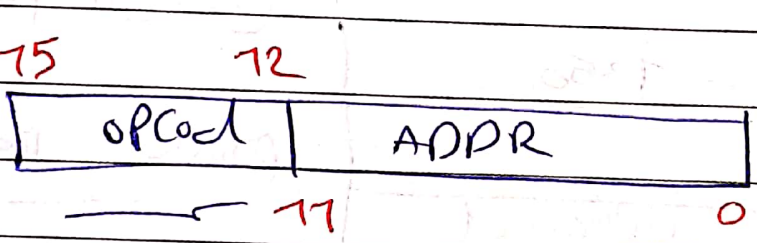
Scanned by CamScanner





داده‌ها

داده‌ها ۱۶ بیت



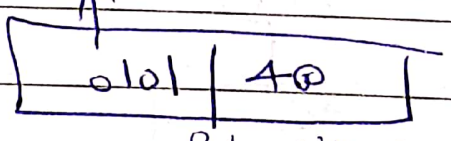
دستورات

۴ بیت برای عملیات نوع

۱۲ بیت آدرس

نوع دستور = FA

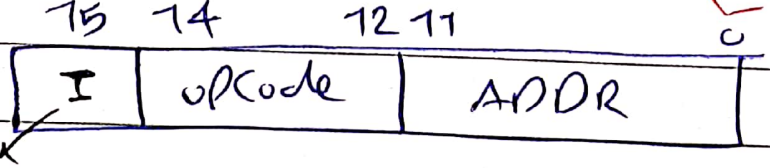
مثلاً یک دستور



مقدار خانه‌ی ۴۰ خونده می‌شود

و میره توی CPU با AC جمع می‌شود و نتیجه توی AC میشه

آدرس‌دهی غیرمستقیم



زیرنگاشت

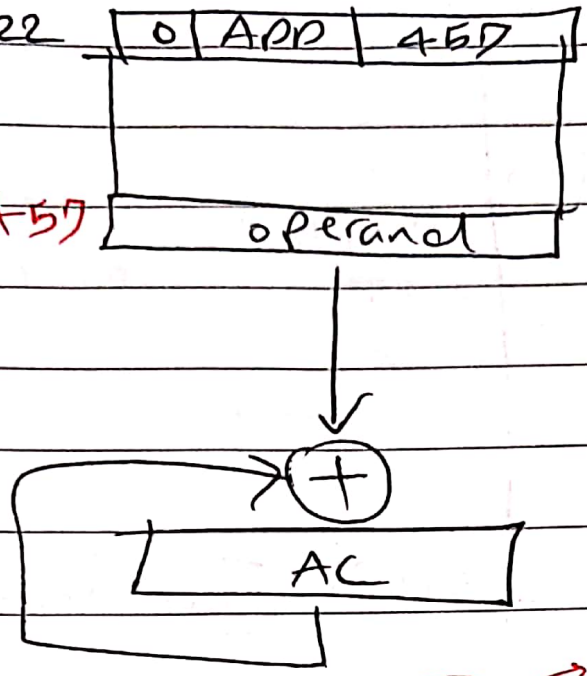
Indirect

اگر = ۱ باشد → آدرس‌دهی غیرمستقیم

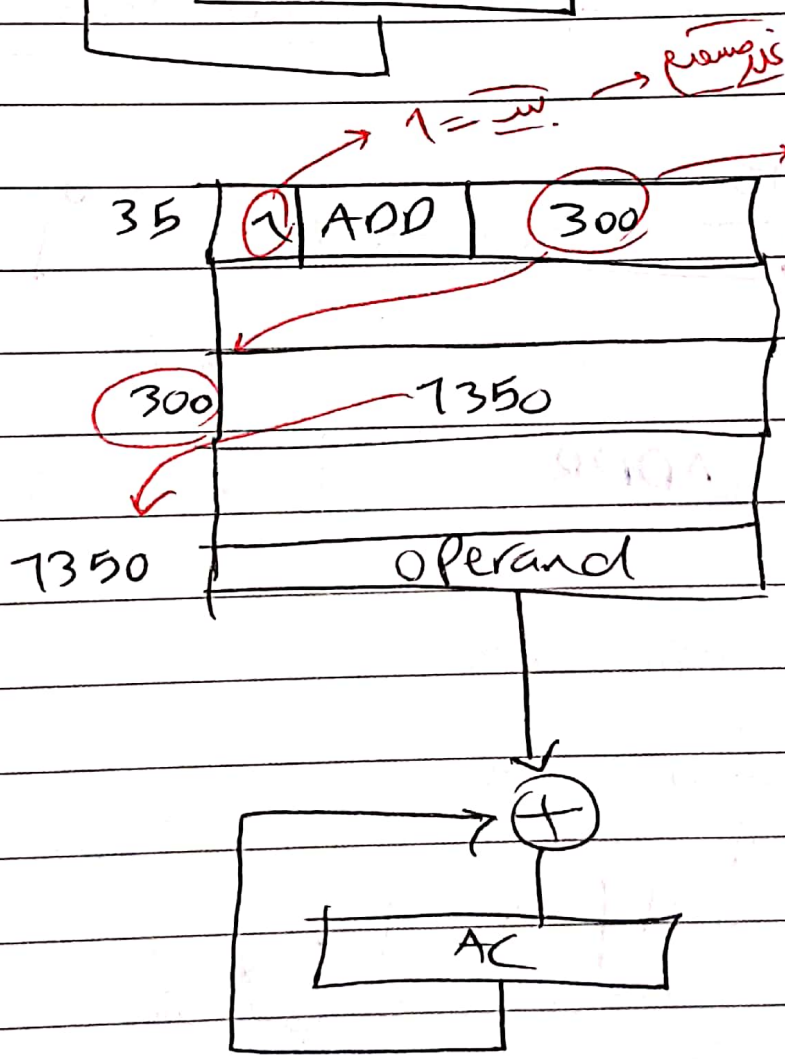
اگر = ۰ باشد → مستقیم

بردار توی خانه‌ی 457 ، 6 مقدار داده‌ی دودویی را بردار با AC

جمع بزن



آدرس دهی مستقیم



بیت = 1 → آدرس غیرمستقیم

آدرس خانه‌ای که باید بریم تا آدرس operand اصلی را بیابیم

مثل pointer

اندازه‌ی عرض حافظه ← پهنای باس

Accumulator

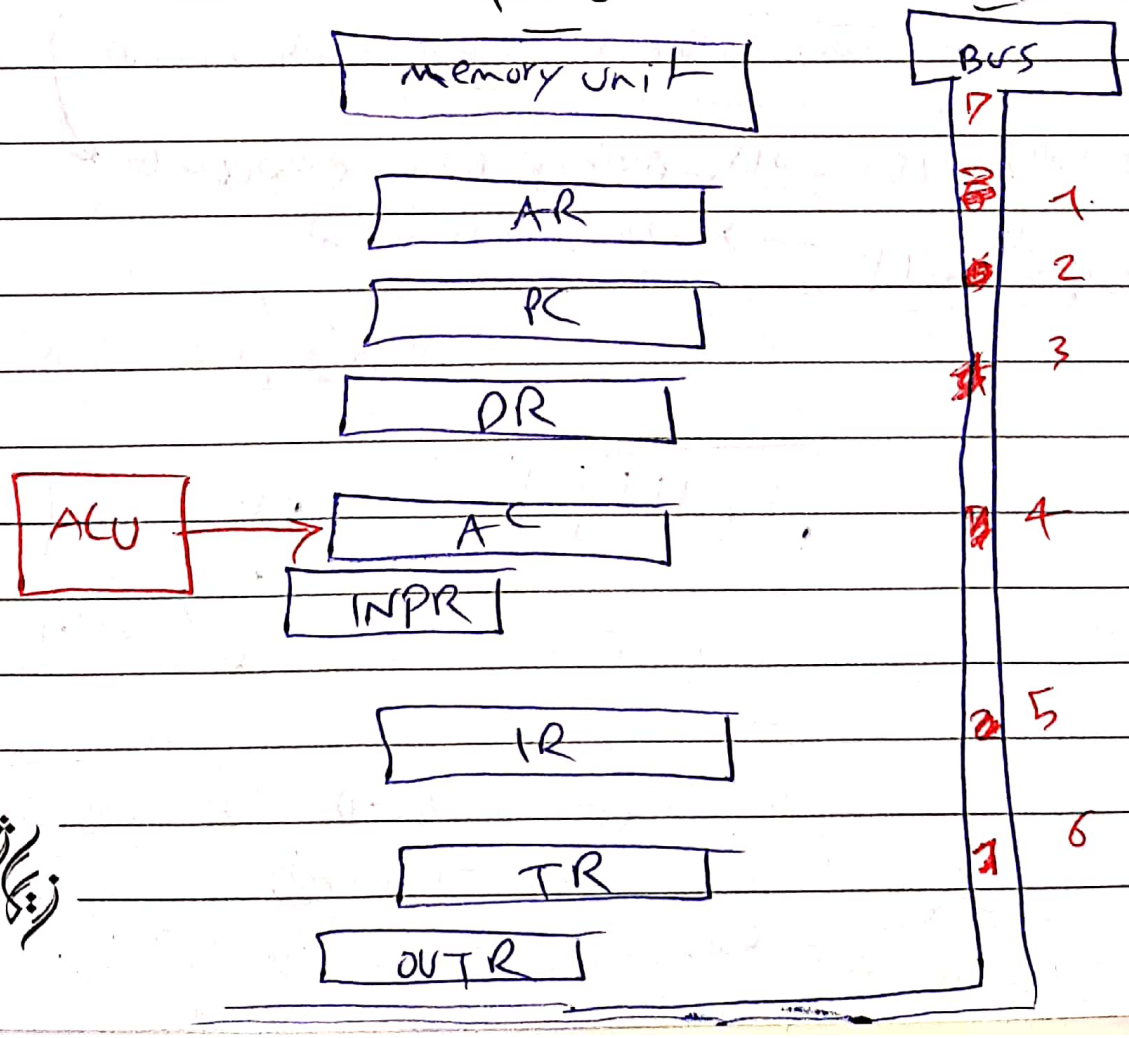
ثباتها	نام	اندازه	پکلمه
	AC	16	ثبات اصلی پردازنده
	DR → Data register	16	ذخیره داده خروجی حافظه
	AR → Address register	12	ذخیره آدرس حافظه
	IR → instruction register	16	ذخیره دستور
	PC → programm counter	12	آدرس دستور
	TR	16	ذخیره داده موقت
	INPR	8	داده ورودی
	OUTR	8	خروجی

شمارنده برنامه

temp register

input register, output register

چرا هست؟ چون ما کدام داریم



Scanned by CamScanner

ورودی AC مستقیماً از ALU می‌گیرد ← تنها اطلاعاتی که از باس ورودی می‌گیرد ← AC است

انواع دستورات:
- memory Ref
- register Ref
- IO Ref

کار با حافظه: کد دستور غیر ۷ است

		آدرس

یک بیت — ۲ بیت

آدرس مستقیم
غیر مستقیم

$2^4 - 2 = 14$ حالت مختلف

با حافظه کاری ندارد ← بیت I به درد نمی‌خورد

کار با ثبات

کد دستور برابر ۷ و بیت I برابر صفر است

حافظه نمی‌خواهیم ← آدرس نمی‌خواهیم ← ۱۲ بیت آدرس بدون استفاده می‌ماند

اما فقط ۱۲ تا دستور داریم

حالت $2^{12} + 6$

کار با IO: کد دستور ۷ و بیت I = یک

۱۲ بیت	۱,۱۱۱

خالی

اما فقط ۶ تا دستور مختلف داریم

دسته‌بندی دستورات:
1. دستورات ریاضی، منطقی و شیفت
2. دستورات انتقال به یا از حافظه
3. دستورات کنترل روند برنامه
4. دستورات ورودی / خروجی

کنترلی
غیر کنترلی

Scanned by CamScanner

Session 10

(carry) بیت نقلی ← E

memory ref. → opcode ≠ 7

Hexadecimal

مجموعه دستورات

Symbol	I=0	I=1	Description
AND	0XXX	8XXX	AND memory word to AC
ADD	1XXX	9XXX	Add ~ ~ ~
LDA	2XXX	AXXX	load ~ ~ ~
STA	3XXX	BXXX	store content of AC in memory
BUN	4XXX	CXXX	Branch unconditionally
BSA	5XXX	DXXX	Branch and save return address
ISZ	6XXX	EXXX	Increment and skip if zero

opcode → بیت → از 000 تا 111 → بنابراین 111 باشد

7 بایستی memory refrence باشد

از صفر تا 6 می‌تواند opcode باشد

CLA	7800	clear AC
CLE	7400	clear E
CMA	7200	Complement AC
CME	7100	~ E
CIR (Rotate)	7080	circulate right AC and E
CIL	7040	~ left ~ ~
INC	7020	Increment AC
SPA	7010	skip next instruction if AC positive
SNA	7008	~ ~ ~ AC negative
SZA	7004	~ ~ ~ AC zero
SZE	7002	~ ~ ~ E is 0
HLT	7001	halt computer

register Ref

اجرای دستورات را متوقف کن

1, 111 → F

INP	F800	Input character to AC
OUT	F400	output " from AC
SKI	F200	skip on input flag
SKO	F100	" " output "
ION	F080	interrupt on
IOF	F040	" " off

I/O

افزاری دیجیتال

لاژیک اما قابل تغییر نیست

واحد کنترل

کند افزاری

قابل تغییر نیست

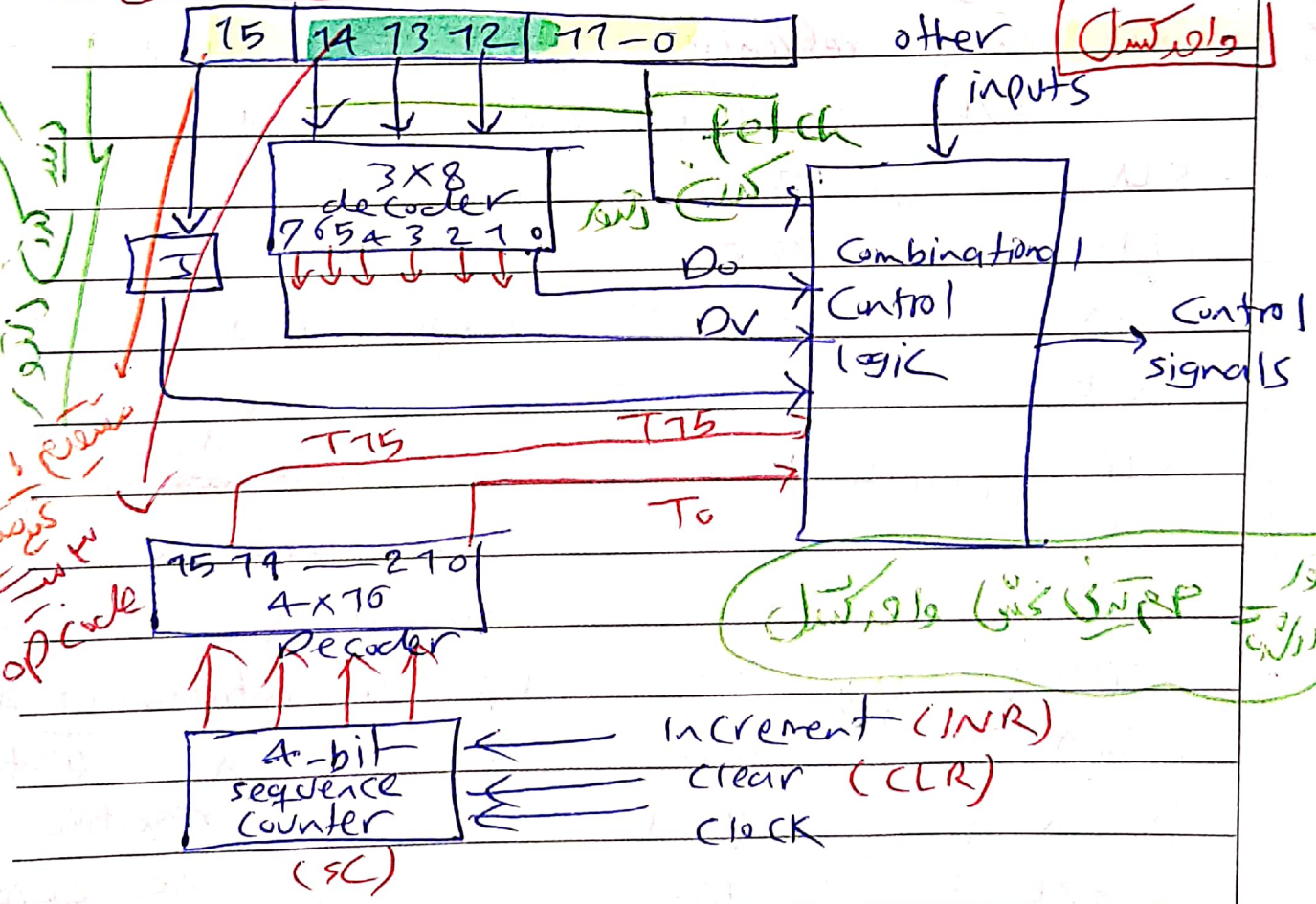
همه کارها با مدارهای مختلف و گیت

ترم افزاری

میکروپروگرام (با تغییر میکروپروگرام میشه دستورات را تغییر داد)

بکاه ترم افزاری

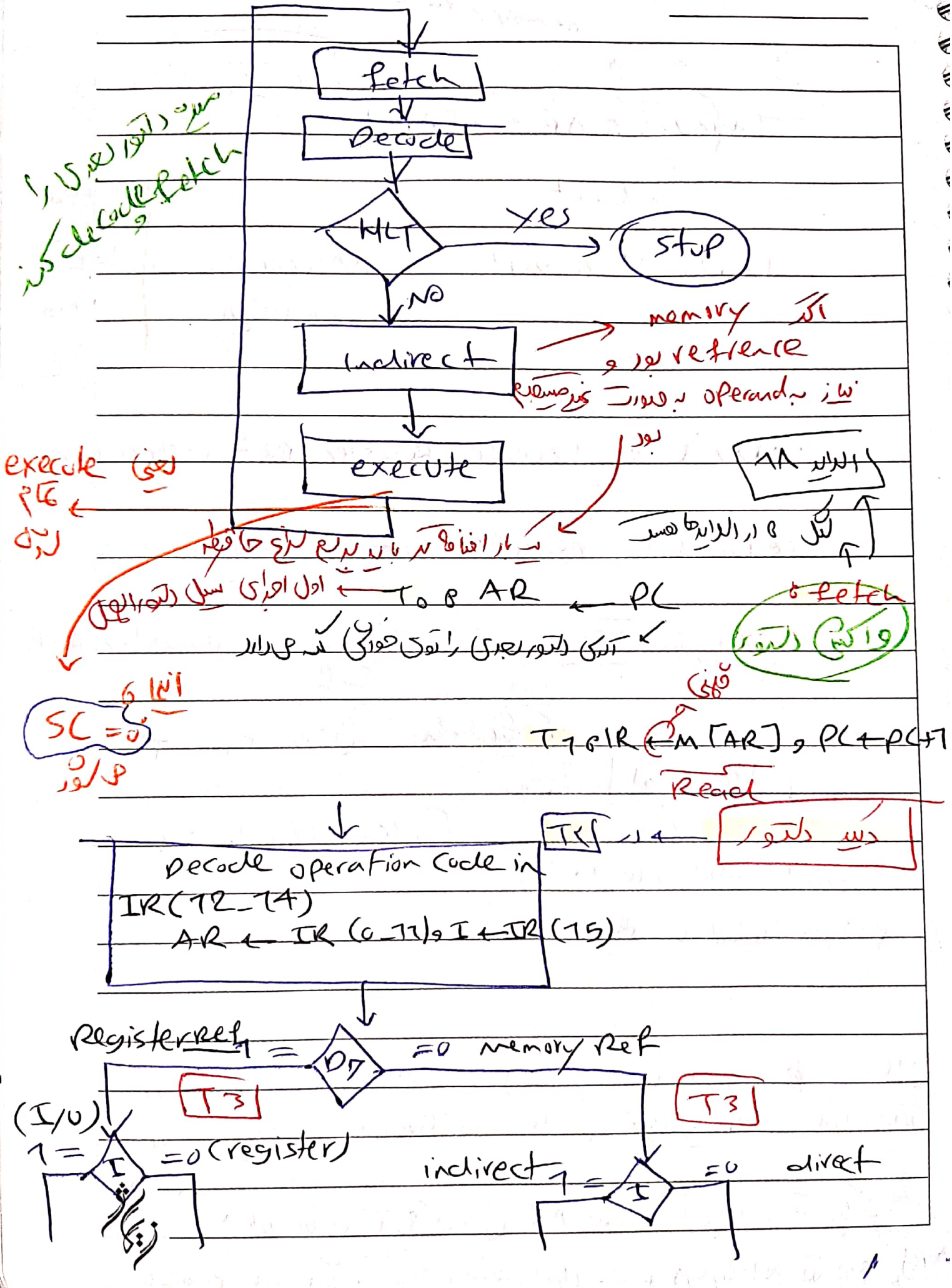
افزایش پیچیدگی و قیمت سیستم



هر وقت SC 0 صفر شود یعنی برو دستور بعدی را fetch کن

زیبا

حداکثر یک دستور 16 بایتی می‌تواند وارد بشه.



Scanned by CamScanner

memory Ref

سیگنال آدرس غیرمستقیم

$T_4 D_7 I$: $AR \leftarrow M[AR]$

Indirect

اجرای memory Ref ها از T_4

$T_4 D_7' I'$: NOP No operation
هیچ کاری نکن

$T_4 D_7 I'$: execute Register Ref instruction

$T_4 D_7 I$: execute I/O Reference

Register Ref و I/O Ref را از T_3 شروع می‌کنیم

fetch برای T_0 , T_1

decode برای T_2

توی T_3

باید بنابر اینکه چه دستوری داریم باید اجرا را شروع کنیم

تمومی همه‌ی تابع‌ها اجرا قابل شروع شدن اند به جز تابع‌هایی که آدرس indirect داریم توی این تابع یک کار اضافه‌تر داریم

باید بیاییم آخر که در AR هست را بخوانیم و آخر کم‌کم خواندیم تازه می‌رسیم آدرس متغیر اصلی ← دوباره باید بیاییم در AR

تا بتوانیم دوباره آدرس را بخوانیم

اگر D7 نشود و indirect فعال بود

یعنی memory ref بود بیت I=1 بود

T4D7'I : AR ← M[AR]

یک کلاک

چیزی که توی AR هست را بخوان بریز توی AR دوباره

لود

آدرس

اگر این لود ها برقرار بود ، اجرای دستور از T4 شروع می‌شود

* برای دستورات memory ref چقی هستند I=0 یا I=1

خاص بستم تصمیم گرفتیم برای اینکه یاد برای I=0 و I=1 ، دو تا مدار ترکیبی جدا ببندند ← تصمیم گرفتیم یک کلاک برای مشخص‌شدن ما (I=0)

اضافه کنند تا اونا هم از T4 اجرایشون را شروع کنند و یک مدار مشترک برای I=0 , I=1 درست می‌کنند

پس memory ref I=0 از T4 شروع به اجرا کنند

I=1

اگر direct باشد

T3D7'I' : NOP

No operation ← هیچ کاری نمی‌کند و اطلاعات قبلی را نگه می‌دارد.

زیگاژ

Scanned by CamScanner

اما اونهایی که I/O و Register ref هستند

از همان T4 شروع به اجرا می‌کنند

T3 D7 I' execute I/O instruction

اجرای دستور

در یک یا چند پالس ساعت دستور اجرا می‌شود

وقتی که دستور تمام شد باید برویم سراغ fetch کردن دستور بعدی

خب خوبی باید برویم سراغ این کار و SC را صفر می‌کنیم

sequence counter

در پایان دستور صفر می‌شود

اجرای دستور کار با بیت ← لحظه‌ها ⑦ I=0 باشد و opcode = 111 ②

T3 باشد ③

$\boxed{D_7 I' T_3 = r}$ → به قرار باشد

لحظه

دستور register ref اجرا می‌شود

I	111	12 بیت

B11 — B1 B0

→ ref

هر کدام معادل 4 بیت آدرس باینری ← همان 12 بیت

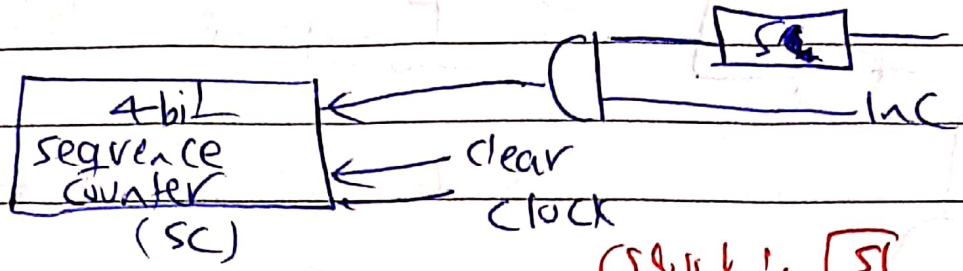
به گفته

دستور اول ← $B_0 = 1$ و بقیه صفر ←

7001 → halt دستور

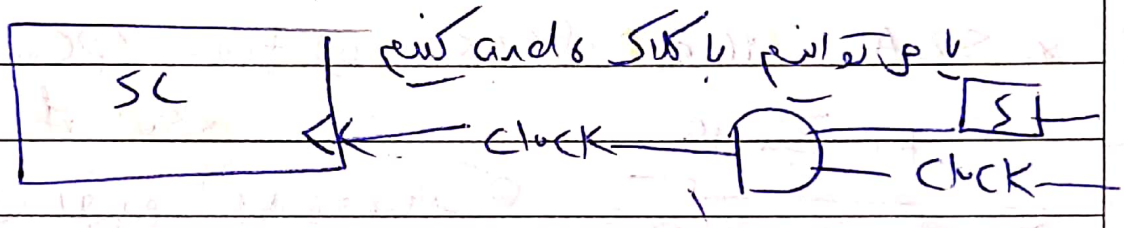
خاموش کردن

برای دستور HLT (halt)



Flag S را با INC and می‌کنیم ← اگر S=1 باشد

وقتی INC فعال است می‌ماند و وقتی S=0 شد ← SC دیگر نمی‌ماند (متوقف می‌شود)



یا می‌توانیم با کلاک and کنیم

مفهوم اجرای دستور یعنی از مرحله دریافت یک که بخصوص که توی حافظه هست و عملیات مورد نظر ما در مت افزار انجام شود

* HLT varb S ← 0 (S is a start stop flip flop)

* SZE vb1 if (E == 0) then (PC ← PC+1)

skip if E zero

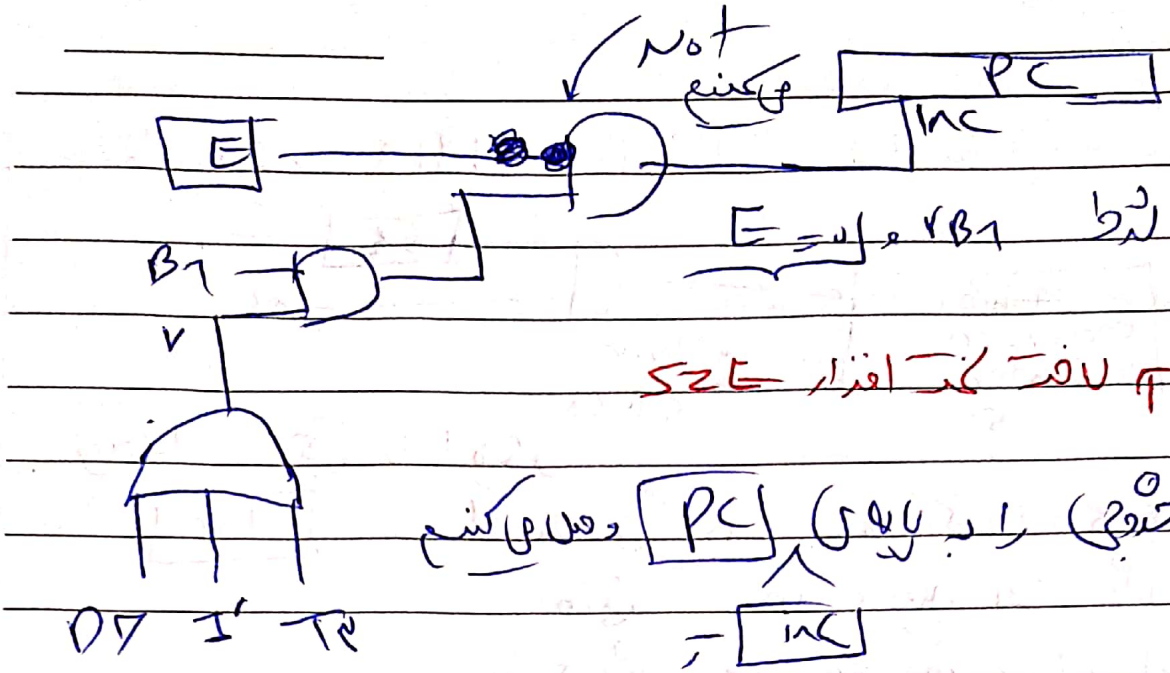
اگر شرط برقرار شد از روی دستور بعدی بپر (skip کن دستور بعدی)

برای پیاده سازی شرط

ما قبلاً توی fetch PC را یک واحد افزایش داده‌ایم حالا یک واحد دیگر هم اضافه می‌کنیم

(دستور بعدی (کنترل شرط)

زیاد



به شرط rB7 و E=0

SZE افزایش کند

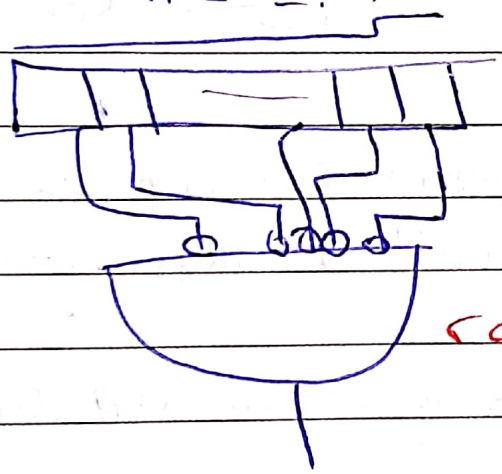
خروجی را به پایه‌ی PC وصل می‌کنیم

INC

* SZA rB2 : if (AC==0) then (PC ← PC+1)

skip if AC zero

AC ۱۶ بیت



صفر بودن AC و مقدار آن

Not gate ۱۶ ورودی

را به یک And بزرگ ۱۶

ورودی می‌دهیم اگر And

یک شد ← یعنی AC=0 است

SNA rB3 : if (AC(15)==1) then (PC ← PC+1)

SPA rB4 : if (AC(15)==0)

skip if positive

skip if negative

INC rB5 : AC ← AC+1 increment AC

به شرط rB5 ، AC یکی زیاد کن

زیبا

* تمام دستورات reg ref توی یک پایه‌ی باشد و کارشون تمام می‌شه ←

Scanned by CamScanner

همیشه وقتی کارهای E انجام میشه باید SC را صفر کند (اول T3 کارش تموم میشه)

E ← AC(0)

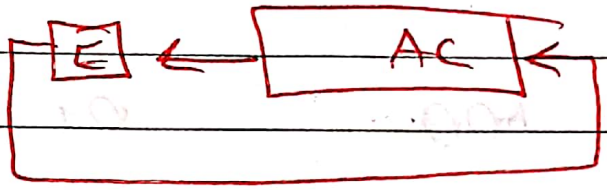
CIR رجیستر B AC ← shr AC , AC(15) ← E ,

CIL B رجیستر B AC ← shl AC , AC(0) ← E E ← AC(0)

, E ← AC(15)

circulate right

circulate left



AC ← AC + 1 و وارد E می‌شه و E وارد AC می‌شه بیت صفر

AC ← AC + 1

بیت carry

AC ← بیت بیت

لیست داره می‌شود

CME B رجیستر B E ← $\overline{E}$ complement E

CMA B رجیستر B AC ← $\overline{AC}$ complement AC

بیت به بیت ، AC را نگات می‌کند (مکمل) و می‌ریزد توی همین AC

CLE B رجیستر B E ← 0 clear E

CLA رجیستر B AC ← 0 clear AC

Scanned by CamScanner

تعداد وليس لاحد صفوف الـ

Memory refrence (التعاملي كله بالحافظة)

عمليات I=0 و I=1 بسبب الـ

Symbol	operation decoder	توضيح
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	$M[AR] \leftarrow M[AR]+1$ if $M[AR]+1=0$ then $PC \leftarrow PC+1$

Scanned by CamScanner

از طریق bus ← مشترک

دستور AND ← از پالس T4 کاری را شروع می‌کند

D_0T_4 : $DR \leftarrow M[AR]$ ①

$T_0, T_1 \rightarrow$ fetch

$T_2 \rightarrow$ decode

$T_3 \rightarrow$ indirect

به این نقطه واحد را شروع می‌کنیم

آدرس پایانی

متغیر را از حافظه می‌خوانیم و می‌ریزیم توی DR و بعد از طریق DR می‌تونه وارد ALU شود

حافظه ← Read

← یک کلاک طول می‌کشد

فعال‌کنی پایه‌های

DR ← load

② اراهی که در پالس بعدی

D_0T_5 : $AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$

حالا متغیری که قرار است با AC و and شود توی DR قرار دارد

DR و AC ، And می‌کنیم

ALU (روی) اتفاق می‌افتد

از طریق ALU و گیت and

کار دستور تمام شد ← باید $SC \leftarrow 0$

دستور ADD

D_1T_4 : $DR \leftarrow M[AR]$ ①

D_1T_5 : $AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$ ②

Scanned by CamScanner

وقتی به HLT رسیدیم ← SC را متوقف می‌کنیم تا دستور بعدی را

چرا باید آخر دستورات HLT بگذاریم؟

کمی اخذ ارش فهمید که

اگر نگذاریم، دستورات می‌آیند SC را صفر می‌کنند یعنی کنترل

تموم شده و میره سراغ دستور بعدی و

چون دستوری نیست بنابراین یک دیتا بخوانند

operand یا

← عملگر الت هی

یا چیز را به هم بزند

باید جلو پایان کار را مشخص

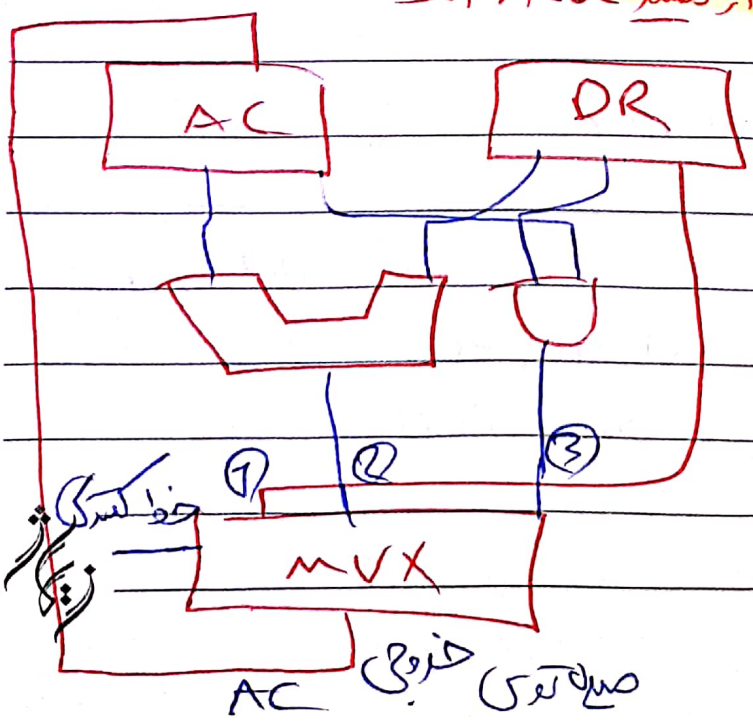
کنیم

دستور LDA DR AC load

① $M[AR] \rightarrow DR$, D_2T_4

D_2T_4 : $AC \leftarrow DR$, $SC \leftarrow 0$

از مسیر ALU الت



صدا ALU ←

سوال: چرا مستقیم M[AR] را نرفتیم توی AC؟

چون AC توی این کامپیوتر به ALU متصل نیست و نمی‌خاهد وارد کار شود.

باید از طریق ALU وارد شود و تنها رجیستری که به ALU متصل است DR است

مهارت بدی است اما لازمه ارزان است

دستور STA از AC می‌بریم توی حافظه

D_3T_4: $M[AR] \leftarrow AC$, $SC \leftarrow 0$ (۶)

AC به باس وصل است و می‌تواند مستقیماً در حافظه بیاید

یک پالس فولادی کشیده فقط

Branch unconditional

دستور BUN

پرش غیرشرطی

مثل goto

خانه 40	BUN 50
PC 47	
50	

BUN آدرس جلوی = 50

یعنی

صبح اینجا

اول که دستور را fetch می‌کند PC را می‌افزاید (قبل از اینکه بفهمد که دستور BUN است یا نه)

اما وقتی معلوم می‌شود دستور BUN است ← با 47 دیگر کاری نداریم

و ادامه برنامه را از خانه 50 حافظه پیگیری می‌کنیم

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

PC را بریز توی آدرس

این چیزی که توی address register هست

برای پیاده‌سازی ساختارهای شرطی:

if { برای کنترل اینکه کدام بخش اجرا شود و کدام اجرا نشود

else { } BUN ← باید وقتی if تمام شد یک BUN بگذاریم که کنترل نره به else

سوال: در زبان‌های سطح بالا می‌گن از goto استفاده نکنیم حتی if و else داریم اما اینجا حتی if و else را خودمان باید پیاده کنیم ← از BUN باید کمک بگیریم

دستور BSA ← برای فراخوانی تابع

branch save address ①

در انتهای تابع باید یک return بگذاریم که بفهمد ادامه‌ی کار را از خانه‌ی بعدی برود

یک جایی باید آدرس برگشت را ذخیره کنیم وگرنه از دستش می‌دهیم

save کنیم ← کجا ذخیره کنیم؟ توی حافظه

جای حافظه؟ ← آدرس برگشت را ذخیره می‌کنیم

خانه‌ی اول تابع را خالی نگه داریم و آدرس برگشت را اونجا ذخیره کن ← دستورات تابع از AR+1 شروع می‌شود

این معماری قابلیت پیاده‌سازی توابع بازگشتی را ندارد

اولین خانه‌ی تابع

لازم دستورات از خانه دوم تابع

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

از خانه دوم لازم کنیم 21 را میدیم توی خانه‌ی اول تابع

② D_5T_5 : $PC \leftarrow AR$, $SC \leftarrow 0$ خانه اول تابع

سوال: تابع را اجرا کردیم چطوری برگردیم به ادامه برنامه؟

با دستور BUN می‌گیم به صورت غیر مستقیم (indirect)

بره خانه‌ی 135 ← آدرس را می‌ذاره توی 135

→ Branch unconditionally می‌کند

→ برمی‌گرده به خانه‌ی 21

Increment and skip if zero (ISZ دستور)

اگر DR که می‌ذاریم توی M[AR] مثلاً صفر شد دستور بعدی skip می‌شود

① D_4T_4 : $DR \leftarrow M[AR]$

② D_4T_5 : $DR \leftarrow DR+1$

③ D_4T_6 : $M[AR] \leftarrow DR$, $SC \leftarrow 0$

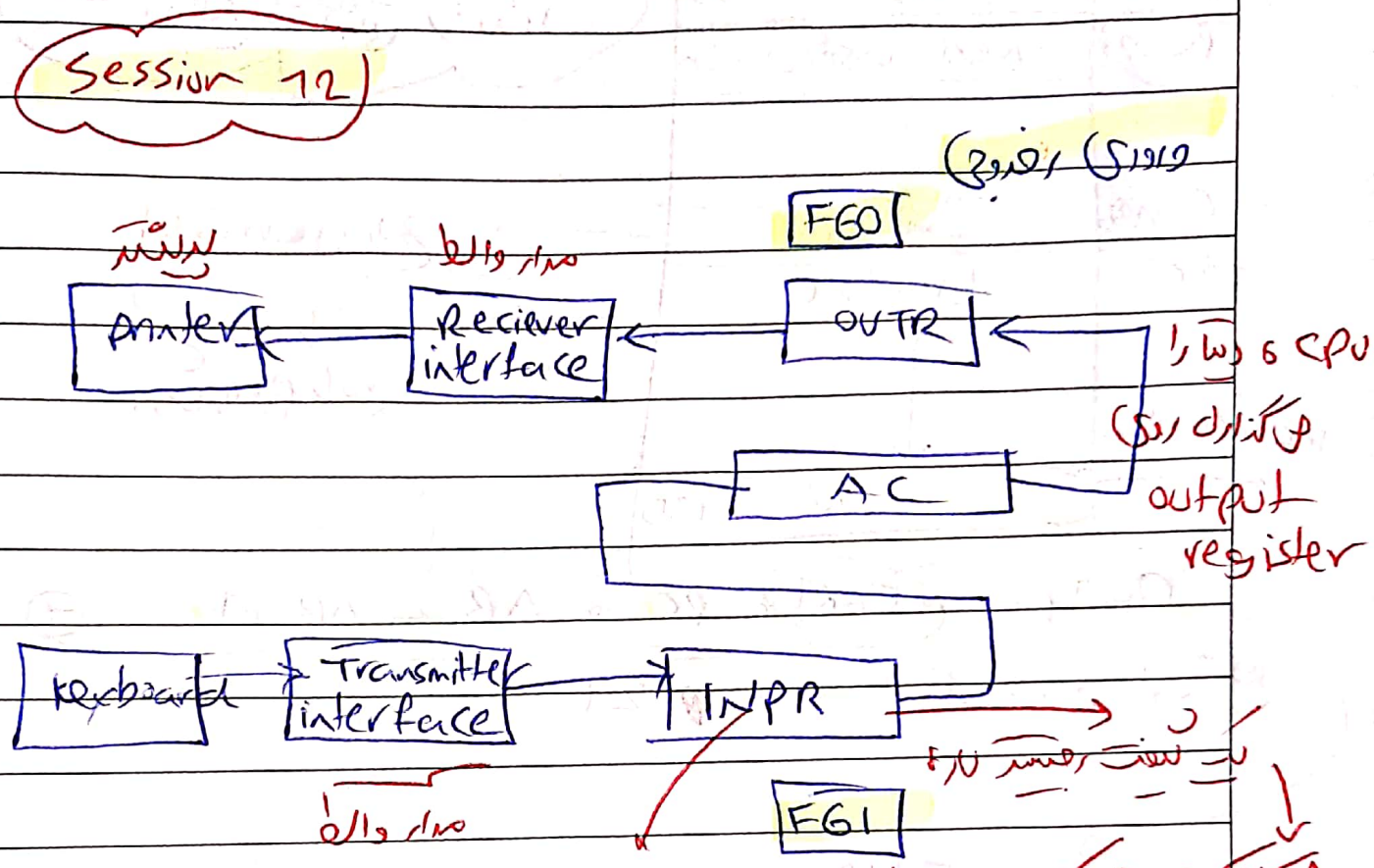
اگر صفر نشد

D_4T_6 ② : $PC \leftarrow PC+1$

Scanned by CamScanner

به چه دستی می‌خوره ه پیاده‌سازی کارتگی حلقه‌ی for

Session 12



نقش مدار واسط ① از نظر الکتریکال ه دستگاه ورودی و خروجی را با CPU هاهنگ می‌کند

یا

② ممکن است دیتای ورودی سریال باشد اما ما احتیاج داریم ۸ بیت پارالل parallel

مدار واسط باید یا بیت یا بیت در ۸ کلاک ه ۸ تا دیتا را دریافت کند و بگذارد توی رجیستر ۸ بیتی و توی یک کلاک بدهد به CPU

حقوقی به CPU خبر بدهد که دیتا ؟ کامل شد

flag input ← FGI

flag ← فعال که بشود ← CPU می‌فهمد که دیتای وجود دارد

CPU که می‌خواهد داده‌ش را (۸ بیت را) توی OUTR بگذارد، فلگ FGO را فعال می‌کند و می‌گه داده آماده است و مدارهای لازم (اگر سریال باشد) پریت بیت به پریسنتر می‌دهد

خروجی

FGI ← در input register

FGO ← در CPU

همه کارشان در فرستنده‌ی اطلاعات در هر بخش است

یک کدام مدارهای واسط برای کنترل کردن دو دستگاه (CPU و keyboard)

کدام کنترل و خواندن را متفاوت می‌کند

یک بافر می‌گذارد

کدر خیلی کم

کدر خیلی بالا

یک صف است (کدر ← OUTR) است

می‌گه CPU که کدرش بالاست، با بهره کدر بالا داده‌ها را بنویس

و ما در Reciever interface آرام آرام شروع به خواندن می‌کنیم

اگر طول صف کم باشد و همین است کدرش بالا، می‌توانیم ادامه صف را در Reciever interface تشکیل دهیم ← یعنی یک بافر بگذاریم (یک آرایه مداوم)

به صورت موازی از OUTR می‌خوانیم ← reg تا ۱۰۰ ← ۸ تایی

توی ورودی چون کدر دریافت پایین است، نیاز به صف نداریم چون اینکه CPU کدرش شلوغ باشد و نتواند به keyboard جواب بدهد (مثل هنگ کردن موس وقتی CPU کدرش شلوغه) ← CPU به دیتاها را گرفته ولی پردازش نکرده تا

حقیقتا صف ←

اجرای دستور کار ورودی/خروجی

$D_7 IT_3 = p($ common to all input output instructions

$\boxed{\text{op code} = 111}$

$\boxed{\text{indirect} = 1}$ ، $\boxed{\text{زمان اجرا} = T_3}$

مثل register refrence

$IR(i) = B_i$ [bit in $IR(6-11)$ that specifies the instruction]

۶ تا دستور داریم ← کدامش را می‌خواهیم با
از B_6 تا B_{11} مشخص می‌کنیم

IOF pB_6 : $IEN \leftarrow 0$ interrupt enable off

بیت هفتم و اگر شرایط بالا را داشتیم
$\boxed{1}$ بود

ION pB_7 : $IEN \leftarrow 1$, interrupt enable on

فعال می‌کند

SKO pB_8 : if $(FGO = 1)$ then $PC \leftarrow PC + 1$

از دستور بعدی بپر

skip on output flag

SKI pB_9 : if $(FGI = 1)$ then $PC \leftarrow PC + 1$

skip on input flag

Scanned by CamScanner

۸ بیت از AC و ۸ بیت کم ارزش

OUT p_{16} , $OUTR \leftarrow AC(0-7)$ و $FGO \leftarrow 0$

AC را بریز توی output register

flag خروجی را صفر می‌کند

INP p_{16} , $AC(0-7) \leftarrow INPR$ و $FGI \leftarrow 0$

و از input می‌ریزه توی ۸ بیت کم ارزش AC

و flag ورودی register را صفر می‌کند

بعد از اجرای هر کدام از این دستورها SC را صفر می‌کنیم

clear SC $SC \leftarrow 0$ p_6

** دستورات I/O و مرجع توی یک کلاک تمام می‌شوند ref

برای T4 مهم نداریم دستور بعدی

انواع اولویت‌های سرویس دهی به ورودی / خروجی

① سرکشی (تند تند چک کردن لوازم) ← تلف کردن وقت

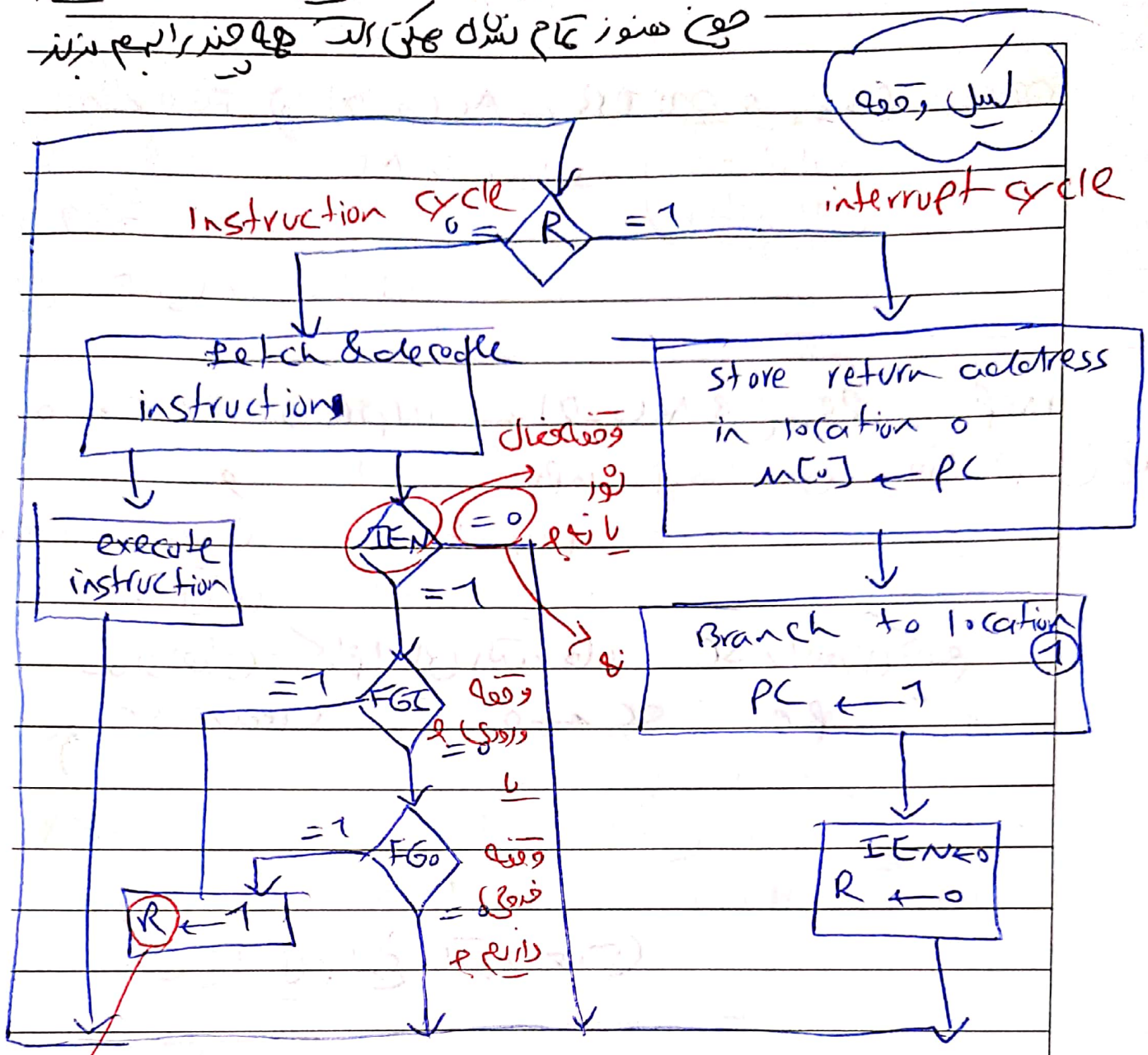
② وقفه (کارمان را متوقف می‌کنیم و به یک کار دیگر می‌پردازیم)

interrupt

معمولاً سرکشی وقت بیشتری تلف می‌کند

سرکشی کی خوبه؟ وقتی یک کاری هم داره تکرار می‌شه (تو چند زمان مشخصی تکرار می‌شود ← باید از سرکشی استفاده کرد) زیرگروه‌های

* توی fetch, decode حق نداریم وقفه را فعال کنیم
چون هنوز تمام نشده ممکن است چند را بهم بزند



قسمت اجرای دستور

چرا نمی‌نویسیم T3 ؟ چون الت

فقط هم الت توی

قسمت وقفه

ورودی/خروجی

لیست وقفه؟

T_0, T_1, T_2 $T_0' T_1' T_2'$ (IEN) (FGI + FGO) R ← 1

(1) (2) (3)

T_0 و T_1 و T_2 نباشد

flag وقفه یک باشد

fetch و decode نباشد

ورودی یا خروجی T6 مادری پردازش دنبال باشد

fetch بعدی میریم لاچ آدرس وقفه

Scanned by CamScanner

آدرس برگشت را می‌ریزیم توی TR

آدرس برگشت را توی خانه‌ی اول حافظه ذخیره می‌کنیم

اگر R فعال شد ←

① $R T_0$: $AR \leftarrow 0$, $TR \leftarrow PC$

② $R T_1$: $M[AR] \leftarrow TR$, $PC \leftarrow 0$

③ $R T_2$: $PC \leftarrow PC+1$, $IEN \leftarrow 0$, $SC \leftarrow 0$, $R \leftarrow 0$

وقفه صفر می‌شه

PC را صفر می‌کنیم

اولین دستور تابع وقفه‌ها توی آدرس (خانه‌ی اول) قرار می‌گیرد ←

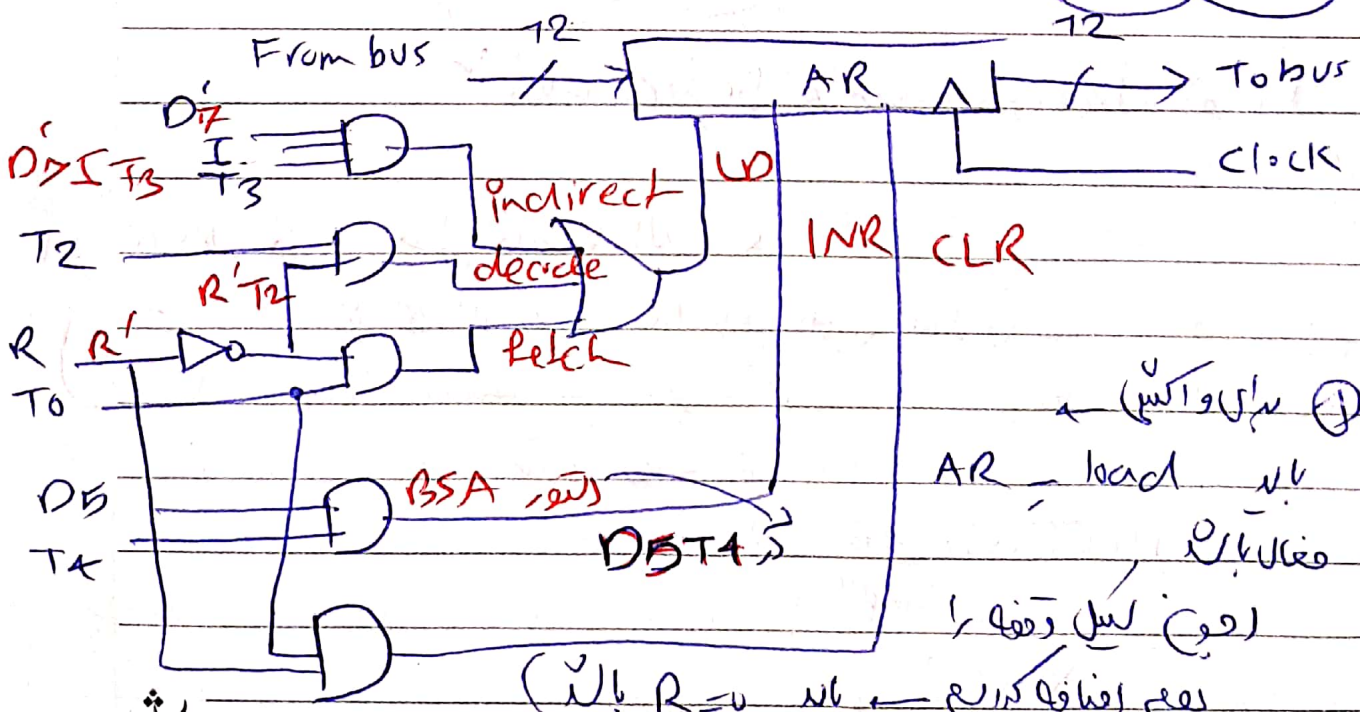
یکی اضافه می‌کنیم

دیگر وقفه نمی‌پذیریم

(وقفه تو در تو نداریم)

در سیکل بعدی T3 ← تابع وقفه‌ها اجرا می‌شود

مدار کنترل AR ← address register



① برای واکشی ← باید AR load فعال باشد

(چون سیکل وقفه را هم اضافه کردیم ← باید R=0 باشد)

$R T_0$ باید باشد

② در T_2 : $AR \leftarrow IR(0-11)$

③ $T_3 D'_7 I$: $AR \leftarrow M[AR]$

Scanned by CamScanner

چون وقفه دارم بیشتر از یک بایت صفر لود می‌کنیم

از 100 لود می‌کنیم (یعنی از صفر تا 100 برای تابع وقفه الت)

برای هفتم

یعنی 256

session 73

برنامه کامپیوتری

① تعدادی دستور المهل متوالی که برای انجام هدفی خاص اجرا می‌شوند

انواع برنامه

- برنامه زبان ماشین یا کدباینری
- برنامه اسمبلی یا مجموعه مدل‌ها، علائم و رقم‌ها تبدیل شود
- برنامه زبان لغو بالا یا نزدیک به زبان معمول

با اسمبلر رانش می‌کنیم تا به صفر و یک

وابسته به کت افزار

و مستقل از کت افزار

کامپایلر → خروجی مستقل الت

* کامپایلر زبان لغو بالا را به اسمبلی تبدیل می‌کند و اسمبلر، زبان اسمبلی را به صفر و یک تبدیل می‌کند.

high level language

OS

روند تبدیل برنامه

HLL (JS, Java, ...)

↓ Compiler

Asmbely

↓ Assembler

Binary code

زبان اسمبلی کامپیوتر پایه

هر دستور از ۳ بخش تشکیل می‌شود:

1. برچسب دستور (و)
2. عملیاتی که قرار است انجام شود
3. توضیحات دستور

اختیاری — باید باشد

Label	instruction	comment

برچسب دستور — دستوری که قرار است انجام شود — اختیاری — اجرا نمی‌شود

(دستورات اصلی زبان اسمبلی) ← با نمت افزار کار دارند

* دستورات کلیدی کامپیوتر مبنا

1. مراجعه به حافظه ← دستور / آدرس عملوند / حرف I
   - غیرمستقیم
   - مستقیم یا
2. مراجعه به ثبات
3. مراجعه به ورودی / خروجی

memory Ref.
register Ref.
IO Ref

Add 500

یعنی AC را با مقداری که در خانه‌ی 500 است جمع کن و در AC بریز

(شبه دستورات زبان اسمبلی)

* برخلاف دستورات اصلی، هیچ کاری با نمت افزار ندارند

این دستورات توسط اسمبلر حذف می‌شوند و معادل باینری در کد زبان ماشین نخواهند داشت

فقط به عنوان راهنما برای اسمبلر استفاده می‌شوند و به اسمبلر می‌گویند که موقع تبدیل کد چه کارهایی را رعایت کند

1. ORG N ← محلی از حافظه که ادامه‌ی برنامه در آن قرار دارد.

Scanned by CamScanner

شروع برنامه باید از خانه‌ی 100 حافظه باشد ← اول برنامه می‌نویسیم ORG 100

مثلاً وقتی می‌نویسیم ORG 100 ← یعنی اولین خانه‌ای که به صفر یک ترجمه کردی را توی خانه‌ی 100 حافظه بریز
(چیزی محتوای ترجمه شده در حافظه)

② END : پایان برنامه به اسمبلر می‌گه که تا کی ترجمه کند
ترجمه نمی‌شود
تا جایی که به END برسد
اما HLT ترجمه می‌شود

③ DEC N : علامت به صورت عدد اعشاری
دهدهی

④ HEX N : علامت به صورت هگزادیمال
هگزادیمال

متغیر که باید بعداً تعریفش کنیم و یک خانه از حافظه را برایش اختصاص می‌دهیم

* برنامه تقریباً دو عدد در حافظه
توی کامپیوتر باید دستوری برای
تفریق نداشتیم ← باید یک کاری
(دستورهای جمع و مثال ۲) به برنامه‌ی
تفریق بنویسیم یک سیگنال است که
ما تعریف کردیم

ORG 100 ← اولین خانه‌ای که مقادیر قیمت افزاری دارد را بگذار در خانه‌ی 100

100 → LDA SUB — 100 — AC اولین متغیر را میبرد توی

101 → CMA // complement AC

102 → INC // increment AC

103 → ADD MIN → عدد اول

104 → STA DIF → چیزی که توی AC هست را بریز توی DIF

105 → HLT

تا اینجا مثال ۲ عدد دوم را در AC داریم
بعد با عدد اول جمع می‌زنیم

106 MIN, DEC 83 → مقدار 23 را
107 SUB, DEC -23 → برای SUB قرار داریم
108 DIF, HEX 0 // difference stored here
تمام شد برنامه کند اجرا END // End of symbolic program

Scanned by CamScanner

* فرض: خانه‌ی 700 باید یک دستور نحت‌افزاری باشد بقیه‌ش به خود برنامه‌نویس بستگی دارد.

HLT را کدی نویسیم برنامه‌ی نحت‌افزار کار می‌شود
← متغیرهایی که توی برنامه تعریف کردیم

اصولاً اول برنامه را می‌نویسند بعد متغیرها (الزامی نیست)

* در وقفه ← آدرس صفر برای آدرس برگشت است و خانه‌ی 1 هم لازم است روال وقفه است

0 و 1 و 700 ← آدرس‌های خاص هستند

ORG 700 → در کامپیوتر پایه ترجمه نمی‌شود

کنترل دیجیتال

(ترجمه به زبان ماشین)

Location	content	Symbolic program
0010 = 2 ← I=0 →	0 010	ORG 700
700	2 707	LDA SUB 707
701	7200	CMA
702	7020	INC
703	706	ADD MIN
704	3708	STA DIF
705	7001	HLT
706	0053	MIN, DEC 83
707	FFE2	SUB, DEC -23
708	0000	DIF, HEX 0
		END

opcode

توی جدول opcode ها نیست

خانه 706

معادل کد هگزا

معادل هگز

Scanned by CamScanner

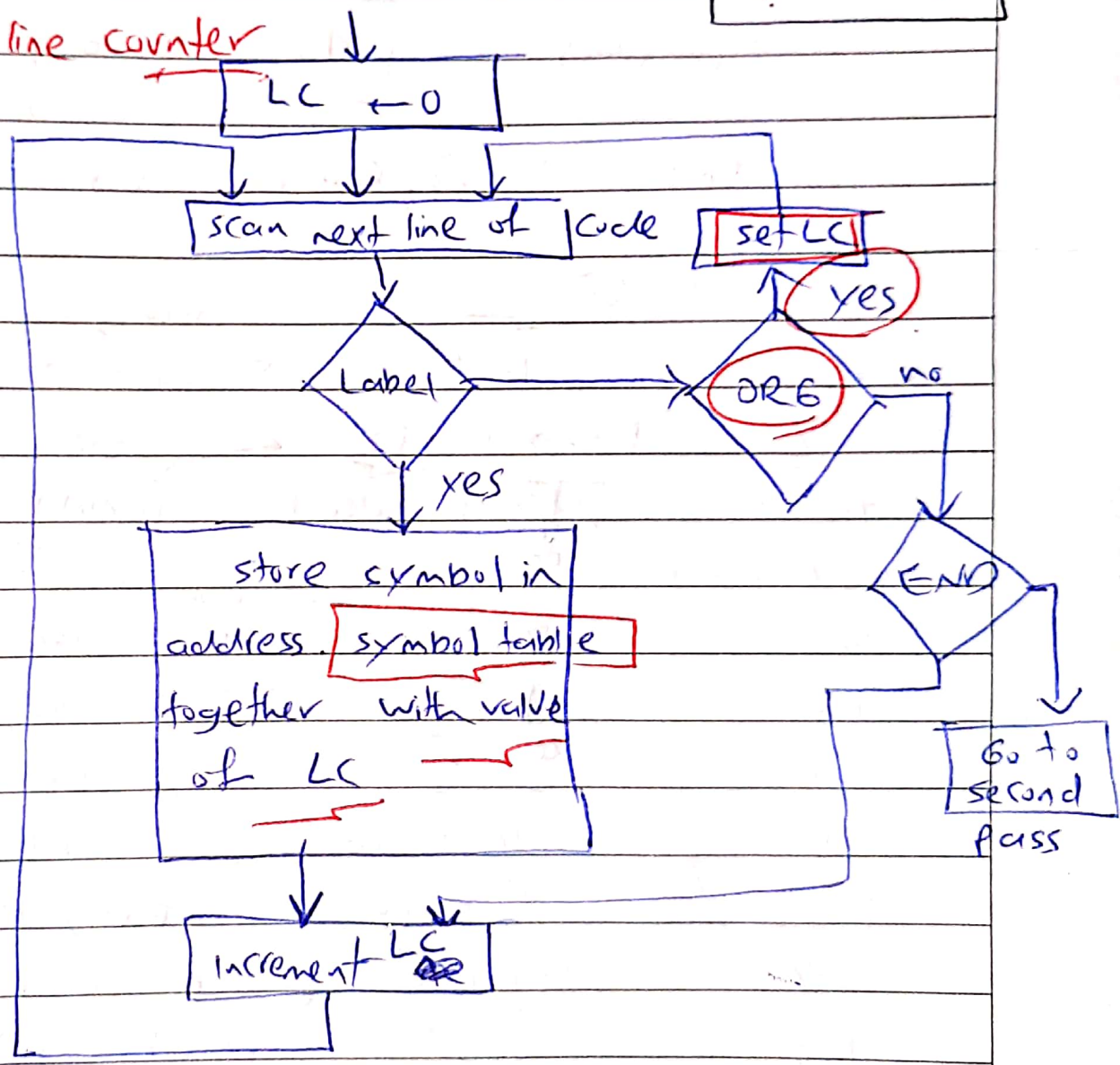
اسمبلر ۵ خط به خط اینها را ترجمه می کند

مراحل ترجمه

① مرحله اول ۵ جدول لیبل ها به همراه آدرس ها بدست میاد

② مرحله دوم ۵ با جدول لیبل ها ۵ ترجمه انجام می شود

مرحله اول ترجمه



اول کار که LC=0 است میرود ORG 700 را می بیند LC=700

زیرا

می شود

LC هم میره جلو تا برسه به دستوراتی که label دارند

705	MOV
707	SUB

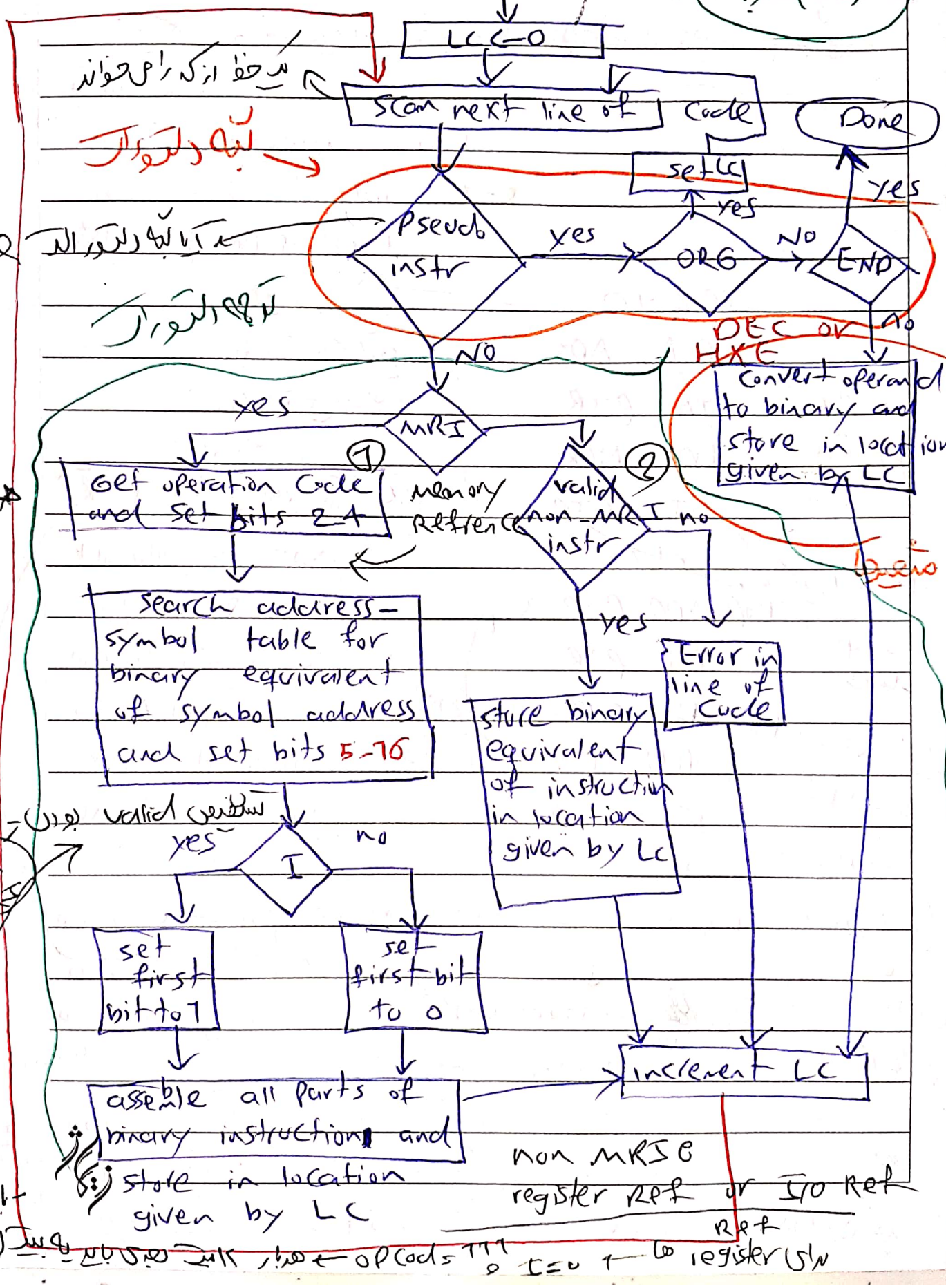
صفر میشه ۵

DIF و معادل 708

line counter را صفر می کند

این کد را انجام می دهد تا به دستور End برسد

مرحله دوم برنامه



Scanned by CamScanner

* اگر memory بود، بیت ۴ و ۵ را برای سیگنال دستوری که به کار بردیم

ref

تعریف می‌کند و بعد صفحه برای اون جدول کمبل‌ها تا ۲ بیت کم ارزش را

پر کند.

اگر مستقیم بود ← I=0

نبود ← I=1

مثال ۵ جمع ۱۰۰ عدد با حلقه

ORG 100	مثل یک
LDA ADS	پوینتر که آدرس شروع
STA PTR	آرایه را توی خودش
LDA NBR	ذخیره کرده
STA CTR	PTR آدرس اولین عدد آرایه را دارد
CLA	باید بدانیم چند بار حلقه باید
Lop, ADD PTR I	تکرار شود؟
ISZ PTR	عملیات در کامپیوتر باید
ISZ CTR	در AC انجام می‌شود
BUN Lop	در ابتدای عملیات
STA SUM	AC را clear می‌کنیم
HLT	

یک کپی از ADS

یک کپی از NBR

شمارنده حلقه

اندازه کانتر را زیاد کن تا به ۰ برسد

صفحه بعدی خوانده می‌شود

اگر به صفر رسید دستور بعدی اجرا می‌شود

وقتی ۱۰۰ بار حلقه تکرار شد

BUN Lop دیگر اجرا نمی‌شود

و سپس دستور بعدی

به صورت کلی مستقیم آدرس در AC وجود دارد را با آدرس در PTR هست جمع می‌زنیم

برای حلقه‌ها

دستور ISZ آدرس خانه‌ای که

عدد داریم را می‌آورد توی DR یکی بیشتر اضافه می‌کند و

در خانه پر می‌گرداند اگر چیزی که اضافه کردیم صفر بود ← از دستور بعدی بپر

مقدار پوینتر را جلو می‌بریم

زیگزاگ

Scanned by CamScanner

* در دستور ISZ ← هیچ کاری با AC نداریم (☺)

متغیرها را هم باید تعریف کنیم

کد کامل برای آرایه

```
ORG 100
LDA ADS
STA PTR
LDA NBR
STA CTR
CLA
LOP, ADD PTR I
     ISZ PTR
     ISZ CTR
     BUN LOP
     STA SUM
     HLT
ADS, HEX 150
PTR, HEX 0
NBR, DEC -100
CTR, HEX 0
SUM, HEX 0
     ORG 150
     DEC 75
     :
     DEC 23
     END
```

ع در 100 در

آدرس 150 ← شروع آرایه از

تعداد عناصر آرایه

کانتر (برای کنترل زمان حلقه)

نتیجه را ذخیره می کند

شروع آرایه از 150

150 را از کجا آوردیم؟

یه برآورد می کنیم می فهمیم که برنامه‌مون بیش‌تر از 80 خط نیست

کد دیگر را می‌دیم صدا زنگ

150 در نظر می‌گیریم

مثال: برنامه اگر از 100 شروع شده باشد و آرایه از 150 شروع شود با برنامه تداخل ندارد

عملیات منطقی

پیاده‌سازی OR برای دو عدد با به کمک AND

مکمل

CMA AND

$A \vee B = \overline{\bar{A} \wedge \bar{B}}$

```
ORG 100
LDA A        عدد A را می‌ریزیم توی AC
CMA          مکمل می‌کنیم
STA TMP      مقدار AC را می‌ریزیم توی TMP
LDA B        عدد B را توی AC می‌ریزیم
CMA          AC را مکمل می‌کنیم
AND TMP      AC را با TMP مکمل می‌کنیم و می‌ریزیم توی AC
CMA          در نهایت AC را مکمل می‌کنیم
STA OR       و در OR ذخیره می‌کنیم
HLT          اتمام برنامه
A, HEX 15    جواب در OR ذخیره می‌شود
B, HEX FOB2
TMP, DEC 0
OR, DEC 0
END
```

تعریف متغیرها

مقدار A = HEX 15

عملیات شیفت

توی کامپیوتر شما افزار و جرخشی باید نقلی داریم شیفت خالی نداریم

شیفت به راست B از کلاک جریک بیت صفر وارد می‌کنیم

E معادل carry

CLE → اول E را صفر می‌کنیم

CIR → بعد شیفت چرخشی می‌دهیم به راست

تنظیم بیت E CLE

CIL

یعنی آخرین بیت 0 شد

تنظیم ریاضی بر روی E اگر عدد مثبت شد از بیت صفر وارد می‌کنیم

اگر منفی شد 0 یک وارد می‌شود

به کمک کدهای زیر

CLE , Clear E to 0

SPA , skip if AC is positive;

CME , AC is negative & set E to 1

CIR , Circulate E and AC

انجام عملیات ریاضی ← اگر در شرط افزایش دستورش بالا که بیشتر در بخش اصلی مورد

باید به ابتدای دستورات موجود برنامه بنویسیم

جمع اعداد بزرگ

جمع دو عدد ۶۴ بیتی حافظه در کامپیوتر پایه 8 ۱۶ بیت

AL
AC

در یک کدام 8 حد اکثر

دو عدد ۱۶ بیتی را می‌توانند جمع بزنند

A | AL, DEC 12 ۱۶ بیت کم ارزش
AH, DEC 32 ۱۶ بیت پر ارزش

BL, DEC 40

BH, DEC 0

Scanned by CamScanner

```
LDA AL      / Load A Low
ADD BL      / add B low, carry in E
STA CL      / store in C Low
CLA         / clear AC
CIL         / circulate to bring into AC(7:0)
ADD AH      / add a high and carry
ADD BH      / add B high
STA CH      / store in C high
HLT
```

AL, _____
AH, _____
BL, _____
BH, _____
CL, _____
CH

محل متغیرها

ضرب دو عدد ← باقیمانده و جمع

حاصل ضرب دو عدد ۸ بیتی در ۱۶ بیت می‌شود

```
ORG 100
LDA Y
CIR
SZE
BUN ONE
BUN ZRO
ONE, LDA X
     ADD P
     STA P
ZRO, CLE
     LDA X
```

CIE

مرحله

اگر E صفر باشد

میبره

وقتی‌هایی که E یک شده

اصلاح دارم

E را باید همیشه صفر کنیم CLE

صفر را

تا مطمئن شویم از کلت رالت صفر میاد

Scanned by CamScanner

```
CIL          یک رقم از x جدا می‌کنیم و ضرب در
STA X        ثبت یافته‌ی n می‌کنیم
ISZ CTR
BUN LOP      * بعد از اولین کار چک کنیم که
HLT          تا جایی برنامه را اجرا کنیم که y ،
             صفر نباشد
X, HEX 0005  مثل دستور SZA
Y, HEX 000B  skip if AC is zero
P, DEC 0
CTR, DEC -8  → LA باید بشمارد
END
```

SESSION 15

تابع: یک قطعه جزء از برنامه که به صورت مکرر تکرار می‌شود

ذخیره آدرس برگشت قبل از شروع اجرای تابع

برگشت به ادامه اجرای برنامه اصلی در انتهای تابع

دستور BSA برای برگشت

Branch save address

دستور کپی از اشغال فضای پیوسته در حافظه

تابع ثبت

```
ORG 100
LDA X
BSA SH4
STA X
LDA Y
BSA SH4
STA Y
HLT
```

Scanned by CamScanner

X, HEX 7234

Y, HEX 4327

شروع تابع

SHA, HEX 0

CIL ; لیست به چپ

CIL ; می‌خواهیم مهمش کنیم از کدام رالت

CIL ; صفر وارد می‌شود

CIL ; ۴ بار کنیم

AND MSK ; راه اول ; قبل از هر CIL ;

BUN SHA I ; یک CLE می‌گذاریم

MSK, HEX FFF0 ; ۴ بیت رالت ۱۲ بیت بیت صفر

END

MSK

به صورت کنیم صحیح بود خط اول تابع و بعد کجا باید بری

تا مقدار E (carry) را صفر کند ← ۴ تا CLE می‌شود

راه دوم ; یک متغیر به اسم MSK تعریف کنیم که ۴ بیت کم ارزش آن صفر باشد ← اگر MSK را با کد به جای آمده And کنیم ← ۴ بیت رالت کد حتماً صفر خواهند بود

تفاوت اولی ; CLE T3 → تابع CLE T4 → EXE = 17

اول و دوم ; AND T5 → یک And → 6

تفاوت ۱۰ کلاک است ← اولی دوم بهتر است.

ارسال پارامتر ورودی به توابع

* یک پارامتر را می‌توان از طریق اکومولاتور انتقال داد
* باید پارامترها را باید از طریق حافظه منتقل کرد

قرارداد → * پارامترها بلافاصله پس از فراخوانی تابع در برنامه اصلی تعیین می‌شوند

تابع OR دو ورودی

* برای هر آرگومانی که از طریق حافظه رد و بدل می‌شود مقدار OR را increment می‌کنیم چرا؟

تا از روی آدرس اول آرگومان (داده) بپرسد بعد کدام دستور بعدی

OR ISZ ← یعنی دستور بعدی به AC خانه‌ی جلوتر را یکی زیاد می‌کند

دلیل ISZ: آیا ممکن است OR صفر شود که از دستور بعدی بپرد؟
خیر آدرس هیچ وقت صفر نمی‌شود داخل برنامه

بزرگترین خانه در حافظه 4095 است ← بزرگترین عددی که ممکن است توی OR بیاد 4095 است که هیچ وقت صفر نمی‌شود

skip می‌کند

```
ORG  200        تابع OR دو ورودی
LDA  X
BSA  OR         → مولفه کمان و داده دارد
HEX  3AF6
STA  Y          → خروجی تابع را می‌اندازد در Y
HLT
```

X, HEX 7B95 → ذخیره آدرس برگشت

Y, HEX 0

OR, HEX 0 — شروع تابع

CMA

STA TMP

LDA OR I → برای گرفتن ورودی آرگومان

CMA

AND TMP

CMA

ISZ OR

BUN OR I

TMP, HEX 0

END

مقدار OR را یک زیاد می‌کنیم تا دفعه بعد برگشتیم

از خانه‌ی بعدی تابع دستورات اجرا شوند

بدون به صورت غیر مستقیم آدرسی که توی خط اول تابع هست را بخوان و مقدارش را بریز توی AC

بدون (return تابع خط اول تابع)

چون تابع را می‌توانستیم طوری تعریف کنیم که هر دو ورودی + خروجی از خانه‌ی حافظه منتقل شود

باید بعد از فراخوانی تابع ۵ تا متغیر تعریف می‌کردیم

تنها ۵ به ازای هر متغیری (آرگومانی) که از خانه‌ی حافظه پاک می‌دهیم باید توی بدنه‌ی تابع ISZ بنداریم ← اگر ۳ تا متغیر پاک می‌دهیم ← ۳ تا ISZ می‌خواهیم

تابع انتقال اطلاعات

می‌خواهیم یک آرایه از آدرس ۶۰۰ را به ۲۰۰ منتقل کنیم (به تعداد عددی که گفته می‌شود)

```
BSA   MVE        برو تابع را بخوان (call function)
HEX   600        آدرس شروع آرایه اول
HEX   200        آدرس شروع آرایه دوم
DEC   16         ۴ تا متغیر
HLT   ---------  اتمام برنامه
MVE, HEX 0       شروع تابع
LDA   MVE I      ---------  لود کردن متغیر اول
STA   PT1        ---------  ذخیره در
ISZ   MVE        ---------  برو خانه بعدی آدرس
LDA   MVE I      ---------  لود کردن متغیر دوم
STA   PT2        ---------  ذخیره در
ISZ   MVE        ---------  برو خانه بعدی آدرس
LDA   MVE I      ---------  ---------
STA   CTR        ---------  ---------
ISZ   MVE        ---------  ---------
LOP, LDA  PT1 I  ---------  برو خانه‌ای که آدرس آن ۶۰۰ است
STA   PT2 I      ---------  را به صورت غیرمستقیم بخوان
ISZ   PT1        PT1 زیاد کردن  و به صورت غیرمستقیم بریز
ISZ   PT2        PT2 ~          توی PT2
ISZ   CTR        CTR ~
BUN   LOP        ---------  تکرار حلقه
BUN   MVE I      ---------  پایان تابع
PT1, ---------   ---------  برو خط اول تابع
PT2, ---------
CTR, ---------
```

خط اول تابع

هر وقت تمام شد حلقه تمام می‌شود

حلقه ۱۶ بار تکرار می‌شود

زیرنویس

Scanned by CamScanner

خواندن از ورودی با کلیدکش

اندر صبر می‌کنیم تا flag=1 شود

```
CIF ,   SKI        / check input flag
        BUN  CIF   / Flag=0, branch to
        INP        / Flag=1, input Check again
        OUT        / print character
        STA  CHR   , store character
        HLT
CHR ,   ____       / store character here
```

با این ورودی را کاراکتر AC

echo

ورودی را توی یه متغیر دیگه ذخیره می‌کنیم تا اگر بعداً خواستیم پردازش کنیم و ____

ارسال به خروجی

```
LDA  CHR   / load character into AC
COF ,  SKO / check input flag
       BUN  COF  / Flag=0, branch to check again
       OUT       / Flag=1 input character
       HLT
CHR, HEX  0057   / character is W
```

اندر صبر می‌کنیم تا خروجی آماده شود

session 78

تابع وقفه قطعی: ① محتوای ثبات‌ها ذخیره شوند

② متناسب با وقفه برخی داده‌ها موردنظر انجام شود

③ ثبات‌ها مقادیر اولیه خود را بازیابند

④ امکان وقفه دوباره فعال شود

⑤ به برنامه اصلی برگردد

Scanned by CamScanner

here

```
ZRO,  ____  /return address stored
      BUN SRV /branch to service routine
      CLA /portion of running program
      ION / Turn on interrupt facility
      LDA X
      ADD Y /interrupt occurs here
      STA Z /program returns here after interrupt
      :
SRV,  STA SAC /store content of AC
      CIR /move E into AC(1)
      STA SE /store content of E
      SKI /check input flag
      BUN NXT / flag is off, check next flag
      INP / Flag is on / input character
      OUT / print character
      STA PTI I /store it in input buffer
      ISZ PTI /increment input pointer
NXT,  SKO /check output flag
      BUN EXT /flag is off, exit
      LDA PTZ I load character from output buffer
      OUT /output character
      ISZ PTZ /increment output pointer
EXT,  LDA SE / restore value of AC(1)
      CIL /shift to E
      LDA SAC / restore content of AC
      ION / Turn interrupt on
      BUN ZRO I /return to running program
```

مثال تابع وقفه

ایجاد وقفه

وقفه رخ داده

وسط دستور

ادامه‌ی کار بعد از جواب به وقفه

ورودی

خروجی

زیگزاگ

Scanned by CamScanner

کنترل میکروپروگرام

وظیفه: اجرای زیرعملیات‌ها بر اساس شرایط و برنامه‌ریزی تعیین شده

واحد کنترل

انواع:
- سخت افزاری
- میکروپروگرام

این قابلیت را به ما می‌دهد تا دستورات پیچیده را تفسیر دهیم و متناسب با نیازی که داریم بدون دست زدن به سخت افزارهای اصلی، ISA را پیشتر را آپدیت کنیم به شکلی که می‌خواهیم.

حقوای سخت افزار را به سمت میکروپروگرام ببریم

instruction set architecture

به جای واحد کنترلی کاملاً سخت افزاری و با گیت، یک حافظه برای واحد کنترل قرار می‌دهیم به اسم حافظه کنترل

توالی اجرای دستورات به کمک حافظه

توالی ریزدستورات را می‌بریم توی حافظه کنترل تا هر دستور خواستیم قرار بدهیم و دستورات جدیدی بسازیم

control memory داخل CPU است

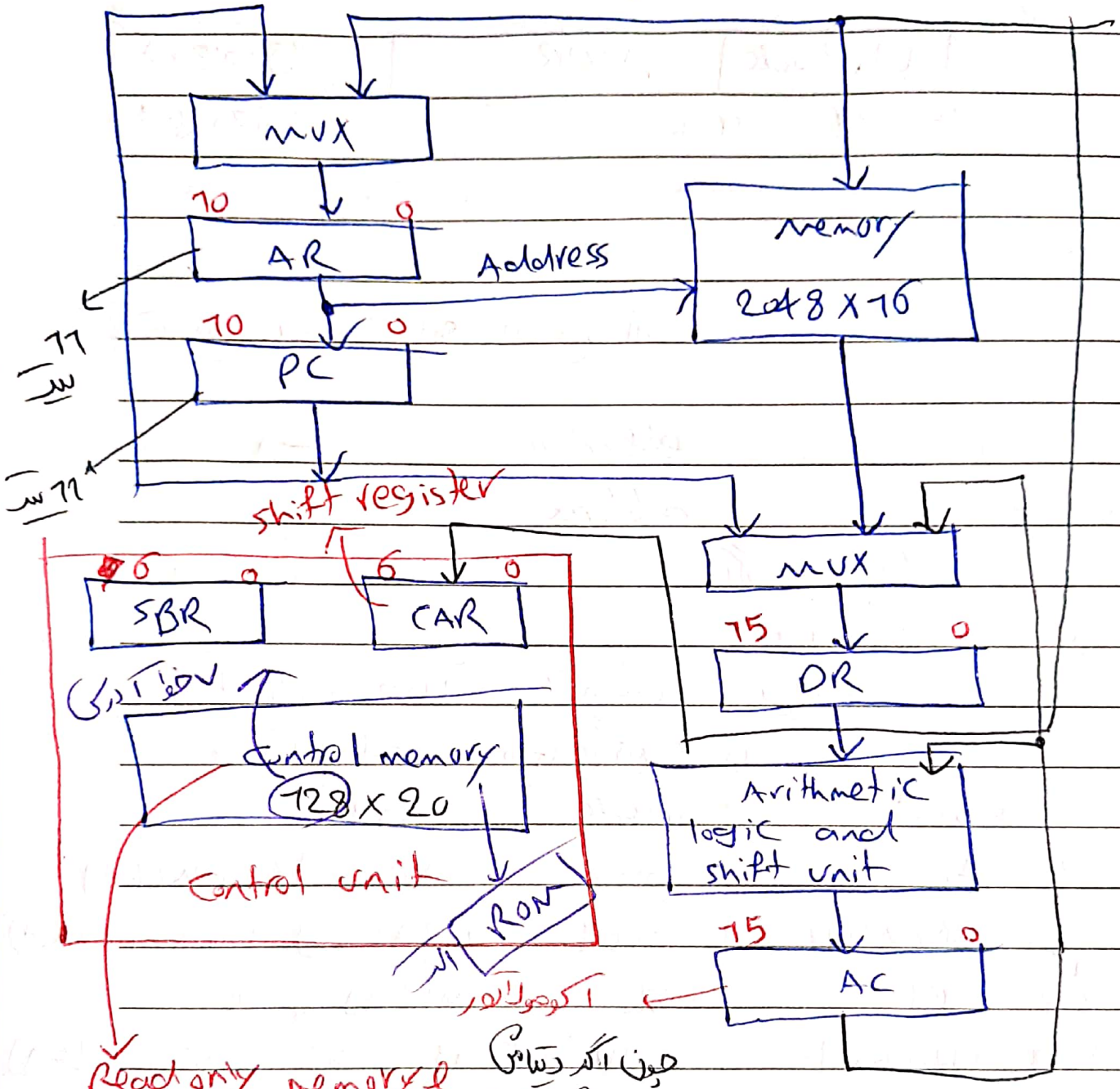
و یک حافظه‌ی جدا است.

Scanned by CamScanner

CAR → control address register

مثل PC می‌ماند — از کجا باید بیاید دستور بعدی

ساختار کامپیوتر با کنترل میکرو پروگرام کجا کند



Read only memory

چون اگر دیتایی پاک شود تا کل کامپیوتر از کار می‌افتد

تغییر

اولین تغییر: حافظه از ۴۰۹۶ به ۲۰۴۸

دومین تغییر: BUS مشترک نداریم

یک مالتی پلکسر

address register

SBR → subroutine → زیرروال

SBR: اجرا کردن یک تابع در حافظه‌ی کنترل ← آدرس برگشت را در SBR ذخیره می‌کنیم

چرا ۴ بیت گذاشتیم تا دستورات بیشتری در اختیار برنامه نویس باشد

قالب دستورات: ۴ بیت opcode و ۱۱ بیت آدرس

I	opcode	ADDR

15 14 11 10 0

که در حافظه اصلی ذخیره می‌شوند

۱۶ بیت

تعریف دستورات پایه به خصوص آدرس

EA ← effective address ← اگر I=0 باشد آدرس همان می‌شود

اگر I=1 باشد ← باید آدرس را به صورت غیرمستقیم بخوانیم

آدرس موثر

اگر indirect بود باید ما آدرسش را در آورده‌ایم ← باز می‌گیم آدرس موثر

Symbol	opcode	RTL
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if(AC<0) then(PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	M[EA] ← AC و AC ← M[EA]

حافظه اصلی

جای عوض کردن آکومولاتور و مکان می‌شود

یک حافظه کش هم داریم با عرض ۲۵ بیت

★ ما در هر دستوری می‌توانیم تا سه آدرس هم زمان انجام دهیم

در یک کلاک ← حداکثر ۲ micro operation هم‌زمان می‌توانیم داشته باشیم

قالب رایز دلتورات

۲۰ بیت

3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

Field	Description
F1, F2, F3	microoperation codes
CD	condition
BR	Branch
AD	Control memory address

Condition (تعیین شرایط)

بپرش بعدی به چه شکلی انجام شود

به چه شکلی، چه پرشی انجام دهیم

در انجام Branch در حافظه کنترلی است

در یک کلاک انجام می‌شود

Branch آدرس ≡ AD

آدرس ۵ بیت است

مقادیر F1

	F1 code (microoperation)	Symbol	RTL
	000	NOP	No operation
	001	ADD	AC ← AC + DR
2	010	CLRAC	AC ← 0
3	011	INCAC	AC ← AC + 1
4	100	DRTAC	AC ← DR
5	101	DRTAR	AR ← DR (10 - 0)
6	110	PCTAR	AR ← PC
7	111	WRITE	M[AR] ← DR

DR TO AC

DR TO AR

Scanned by CamScanner

مقادیر F2

~~Symbol~~

	F2 Code	symbol	RTL
0	000	NOP	No operation
1	001	SUB	AC ← AC − DR
2	010	OR	AC ← AC ∨ DR
3	011	AND	AC ← AC ∧ DR
4	100	READ	DR ← M[AR]
5	101	ACTDR	DR ← AC
6	110	INCDR	DR ← DR+1
7	111	PCTDR	$DR_{(10-0)}$ ← PC

مقادیر F3

F3 code	symbol	RTL
000	NOP	No operation
001	XOR	AC ← AC ⊕ DR
010 complement	COM	AC ← $\overline{AC}$
011	SHL	AC ← shl AC
100 shift به چپ یا راست	SHR	AC ← shr AC
101	INCPC	PC ← PC+1
110	ARTPC	PC ← AR
111	Reserved	

* تو کار یک دستور می‌توانیم بنویسیم که مثلاً F2 و COM انجام دهد

F2 و SUB انجام دهد خیلی چیزی مقصد هم در AC می‌شود

و ما می‌توانیم توی یک پالس ساعت F2 و F3

با مقصد یکسان انجام دهیم.

micro operation

پس انتخاب از جدول ها، عدد آن باید چک کنیم با هم تلاقی نداشته باشد

مقادیر CD ← بیت Condition

چک کردن لدوهای مختلف

CD	Symbol	Condition	Description
00	U	—	Always
01	indirect I	DR(15)	if indirect
10	Signed S	AC(15)	if negative
11	zero Z	AC = 0	if zero

اگر AC(15) منفی بود دستور بعدی را انجام بده

اگر دستور indirect بود branch بعدی را انجام بده

اگر AC=0 بود شرط بعدی را انجام بده

مثل قبل

PC+1 می‌شود

جایگاه بود دستور بعدی

مقادیر BR

BR	Symbol	RTL	Description
00	JMP	CAR ← AD	true
		CAR ← CAR+1	false
01	CALL	CAR ← AD, SBR ← CAR+1	if condition is true
		CAR ← CAR+1	false
10	RET	CAR ← SBR	if negative
11	MAP	CAR(5-2) ← DR(14-11)	if zero
		CAR(1,0,3,0) ← 0	

آدرس مقصد را می‌گیره توی SBR که بعداً بتوانیم به این آدرس return کنیم.

آدرس برگشت بودن

Scanned by CamScanner

با نمایش

MAP یعنی از opcode بفهمیم کجای حافظه‌ی کنترل را می‌خواهیم
اجرا کنیم ← به این کار map می‌گوییم

این ۴ بیت الت ولی آدرس حافظه‌ی کنترل ۷ بیت الت

باید به طوری این ۴ بیت را نمایش کنیم به ۷ بیتی

① یک صفر بهشون می‌گذاریم
② دو تا صفر بهشون می‌گذاریم ← ۴ بیت می‌شه ۷ بیت

BRANCH میگرد که مربوط به

opcode = 0001 → 0 0001 00
opcode

STORE → opcode = 0010 → (در خانه‌ی ۴)
0 0010 00
opcode

در خانه‌ی ۸ انجام می‌شود

EXCHANGE → 72 (در خانه‌ی) →
دستورات ۴۰ تا ۴۷ در حافظه‌ی کنترل چیده می‌شوند
MAP طبق

تا بتوانیم ادامه‌ی کار را از دستور fetch شده انجام دهیم

چرا؟

fetch

① AR ← PC
② DR ← M[AR], PC ← PC+1
AR ← DR(10_0), CAR(5_2) ← DR(14_11),
CAR(1,0,0) ← 0 ← ۴ بیت = opcode → MAP می‌کنیم

(واکشی، دید دستور)

① decode = آدرس = ۱۱ بیت

Scanned by CamScanner

برای fetch طبق منطق پیش ←

① AR ← PC معادل ← PCTAR

تا وقتی آدرس در کلاک قبل load نشده باشد ← نمی‌توانیم read کنیم ،
PC را هم نمی‌توانیم اضافه کنیم ← فعلاً کاری نمی‌توانیم
بکنیم ← یعنی F2 و F3 را NOP می‌گذاریم

اما برای خلاصه نویسی ، NOP نداریم

در لیست بعدی میریم دستور بعدی از حافظه کنترل را می‌خوانیم
باید بگیریم ، بدون هیچ شرطی ، پرش کن به آدرس بعدی

NEXT (یک سیگنال برای آدرس بعدی) JMP U

PCTAR U JMP NEXT

سپس ، محتوی مورد نظر را در دیتا رجیستر می‌ریزیم ← read می‌کنیم

MAP کنیم

F2 = 100 → READ

و PC را هم اضافه کن ←

F3 = 101 → INCPC

READ , INCPC U JMP NEXT

برای دستور بعدی (دیکد کردن) ، DR را بریز توی AR
DRTAR
ما دستور را آوردیم (fetch کردیم ، حالا باید
decode کنیم ، به بالا نگاه این خانه‌ای از حافظه که opcode در DR مشخص کردن را
پژ اجرا کنیم

خودش آدرس را تولید می‌کند (با اضافه کردن صفر به چپ و دو تا صفر به راستش)

ORTAR و MAP

پس 8 fetch

ORG 64

FETCH: PCTAR و JMP NEXT

64 حفظ: READ, INCPC و JMP NEXT

65 حفظ: ORTAR و MAP

66 حفظ

ADD دستور ← MAP ← 0 آدرس صفر

BRANCH دستور ← MAP ← 4 〃

STORE 〃 〃 ← 8 〃

EXCHANGE 〃 〃 ← 12 〃

لازم از آدرس 64 در حافظه‌ی کنترلی

4 بیت opcode داریم ← چند تا دستور می‌توانیم داشته باشیم؟ 4 ← 16

از صفر تا 63 ← به دستورات مربوط می‌شود

آخرین دستور مربوط به اینا در خانه‌ی 63 قرار می‌گیرد

0 B ADD

4 B BRANCH

8 B STORE

12 B EXCHANGE

می‌توانیم fetch را بگذاریم توی خانه‌ی 64

از 64 تا آخر آرایه

64 تا 127 B

از 63 برو

MAP ← معادل همان Decode است

پس کامپیوتر جدید که روشن می‌شود CAR باید عدد باشد ۶۴ باید شود

تا اولین کاری که می‌کند fetch کردن باشد

حافظه کنترل ← میکرو کد دارد ← یعنی کدهای کوچک کوچک می‌کند

دستوراتی که توی کامپیوتر پایه بشوند register refresh می‌گفتیم

را اینجا چند تا تو پشت سر هم قرار می‌دهیم

که مثلاً که دستور ADD ← یک میکرو پروگرام می‌سازیم

را انجام می‌دهد "کنترل"

کم کم شبیه زبان افزاری شد

چرا ۸ تا بتوانیم سیستم را اکیپت کنیم و دستورات جدید بعدی اضافه کنیم

برای واکنش دستورها

F3

110,000,000,00,00 1000001

F1 F2 CD BR

آدرس = ۶۵

Ctrl mem address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

دستوری که در خانه‌ی ۶۴ قرار گرفته

NOP

READ

INCPC

JMP

مهم نیست

65

66

MAP = 11

به 66 باید اشاره کند

به 65 باید اشاره کند

Scanned by CamScanner

وقتی MAP انجام می‌دهیم آدرس اراده‌ای که کجا ذخیره می‌شود؟
طبق خود دستور MAP ← خودش از دستوری که fetch شده است
آدرس جدید را پیدا می‌کند ← ۴ بیت از دستور fetch شده را
توی CAR(15-2) می‌ریزد و بقیه‌ی بیت‌های CAR را هم صفر می‌کند

↓

این دستوری که می‌خاهیم متناسب با عملگر و آدرس پیدا می‌شود

↓

نیاز نیست جدولی دستوری بسازیم

→ 0000000 در آخر کار فقط برای برگشتن به بیت است
اگر ۴ آدرس بعدی از های دستور fetch شده بدست میاد

control ← CAR ← توالی یک دستورات را جلو می‌برد
اصلی ← PC ← توالی دستورات اصلی را

محاسبه آدرس غیر مستقیم ← مرحله بعد

DR ← M[AR] READ
AR ← DR(10-0) DRTAR

* همه دستوراتی که در اختیار میکرو پروگرام داریم با حافظه کار دارن همشون

register ها کجا رفتن؟ اونها توی F_1, F_2, F_3 هستند
باید یک کد بنویسیم که سیگنال indirect را انجام دهد
برای اینکه به ازای هر دستور ↑ ای که داشته باشیم ←
حتی داریم حافظه اشغال می‌کنیم تابع میاریم می‌نویسیم

۴۴ تا ۶۶ را برای fetch, map کنار گذاشتیم

از ۶۴ به بعد نیاز نیست چون خودمون تنظیم می‌کنیم

ORG 67

Indirect, READ → یک کدی باید صدا کنیم اگر نه لود تمام می‌شه

می‌گیم بدون هیچ شرطی برو کن به خط بعدی که ↓ دستورهای کار از خط بعدی

ORG 67

Indirect, READ U JMP NEXT

DRTAR U RET [آدرس]

بدون هیچ شرطی — نمیخاد

برگرد به همان جایی که دستور کارت را انجام می‌دادی

از این باید برگردیم به جایی که صدا زدل

با خط‌خوانی این در هر دستوری می‌توانیم از کی برای عملیات indirect استفاده کنیم

صرفه‌جویی در حافظه

دستور ADD ← opcode = 0000 ← در حافظه‌ای از صفر به بعد کدام پیاده‌سازی می‌شود [0]

اول صفر یعنی indirect را خط‌خوانی می‌کنیم ORG 0

ADD NOP I CALL INDIRECT

label

تا effective address را بدست بیاریم

این خط‌ها کافی

کار خاصی انجام نده فقط برو در صورتی که [I] فعال است برو خط‌خوانی کن INDIRECT را

Scanned by CamScanner

آدرس

Call عمل می‌کند: قبل از اینکه برنامه به حلقه جدید، آدرس خط بعدی را
می‌رود توی رجیستر SBR
تا وقتی که return می‌کند ادامه‌ی کار از خط بعدی اجرا شود
↓
فرقش با JMP: آدرس برگشت را ذخیره می‌کند

لوگوی چیه I ← Indirect باشد
بیت ۱۶ دستوری که پر شده
۲ بیت شرط → CR

اگر I فعال نبود یعنی Condition false بود ←
میره دستور بعدی ⟶ CAR ← CAR + 1
یعنی
اجرا می‌شه

پس خط روبرو: ADD, NOP I CALL I INDIRECT
زمانی اجرا می‌شود که آدرس indirect باشد اگر نباشد میره دستور
بعدی. حالا می‌خوایم M[EA] ← AC
پس اول باید مقدار EA در حافظه را بخوانیم ←
تا تمام نشده باشه نمی‌تونیم کاری کنیم ← READ می‌کنیم
↓ ادامه‌ی کار در دستور بعدی
یک کلاک می‌خواهد

حافظه‌ی جمع بزنیم: F1 = DR ← عملیات جمع
AC ← AC + DR
در آخر باید حافظه‌ی روال خواندن
از حافظه‌ی میکرو پروگرام را تغییر
دهیم که بریم سر fetch
ADD و JMP FETCH

مهاری

Session 17 | اراطی

دستور ADD

		OR60			
0	ADD P	NOP	I	CALL	INDRCT
1		READ	U	JMP	NEXT
2		ADD	U	JMP	FETCH

پس آدرس 3 چی میشه؟

خالی می‌موند

دستور بعدی از 4 شروع میشود.

اگر تعداد دستورها بیشتر 4 تا شد چیکار کنیم؟

به کمک JMP ارجاع می‌دهیم به یک آدرس دیگر

۴۴ و ۴۵ و ۴۶ برای fetch بود

۴۷ و ۴۸ هم برای indirect الت

از ۴۹ تا ۱۲۷ آزاد الت ← می‌توانیم یک JMP بزنیم و اراطی دستور را اونجا پیاده سازی کنیم.

دستور BRANCH

opCode = 0006

آدرس موند

if (AC<0) then PC←EA

مثل SNA الت

در چه خانه‌ای از حافظه قرار می‌گیرد؟ 4 | 0006 00

به شرط 5 نگاه می‌کنیم ← اگر شرط 5 برقرار الت یعنی AC(15) منفی الت ← یک کاری انجام بدیم

Branch 0 NOP 5 JMP OVER

اگر شرط 5 برقرار الت جامپ کن به over

اگر برقرار نباشد چی؟ مسیر خط بعدی ← حالا چیکار باید بکنیم؟

باید بریم دستور بعدی را fetch کنیم

NOP U JMP FETCH

Scanned by CamScanner

OVER 8 NOP I CALL INDRCT

برو آدرس موثر را با فراخوانی indirect بدست بیار

و

EA که توی AR الان رو میریزه توی PC

ARTPC U JMP FETCH

که نیاز 8

ORG 4

BRANCH8 NOP S JMP OVER

NOP U JMP FETCH

OVER 8 NOP I CALL INDRCT

ARTPC U JMP FETCH

آیا می‌توانیم این دستور را ببریم بالا و به عنوان دستور اول اجرا کنیم؟

آره می‌توانیم

اما باعث می‌شود اگر این دستوری که خ کردیم ، لزوماً S را نداشته باشه ، اگر اومدیم دستور indirect بودن رو بررسی کردیم و یک کلاک هدر صرف داگر S بوده باشد)

این خط فقط روی AR کار می‌کند و دست به AC نمی‌زند

دستور STORE ← opcode ← 00010

0 00010 00 → 8 آدرس

M[EA] ← AC

ORG 8

STORE 8 NOP I CALL INDRCT

برای اینکه بتوانیم AC را بریزیم توی حافظه 8 مستقیم راهی نداریم

اول باید بریزیم توی DR و بعد DR را بریزیم توی حافظه مورد نظر

ORG 8

STORE, NOP I CALL INDRCT

ACTOR U JMP NEXT

WRITE

دستور EXCHANGE

ORG 72

EXCH, NOP I CALL INDRCT ① آدرس دستور را پیدا می‌کنیم

READ U JMP NEXT

ACTOR, ORTAC U JMP NEXT

WRITE U JMP FETCH

② Read می‌کنیم ← عوض را می‌خوانیم میاریم توی DR

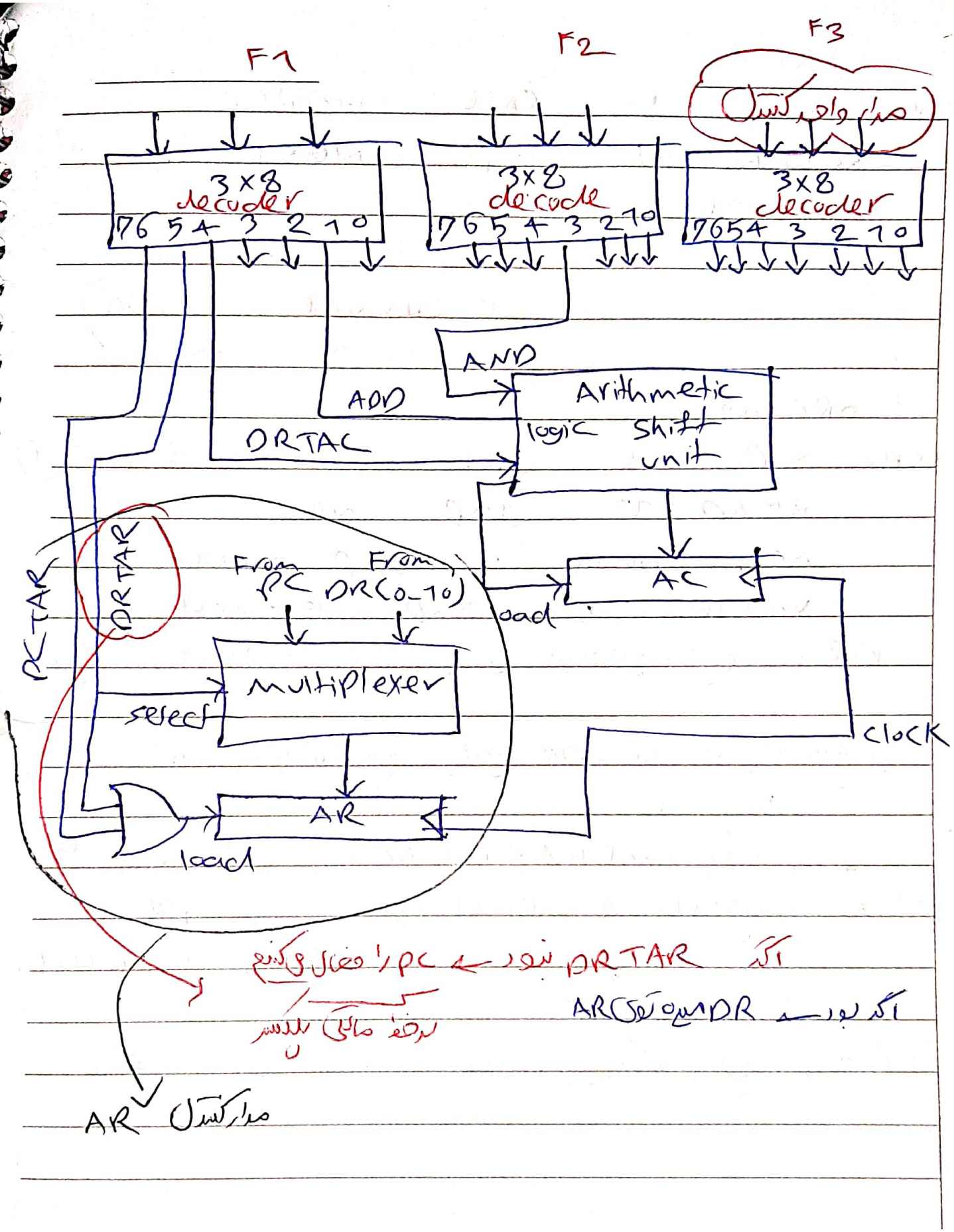
③ در کلاک بعدی می‌توانیم هم زمان DR و AC را با هم جابجا کنیم

④ در کلاک آخر AC را با M[EA] جابجا کنیم ←

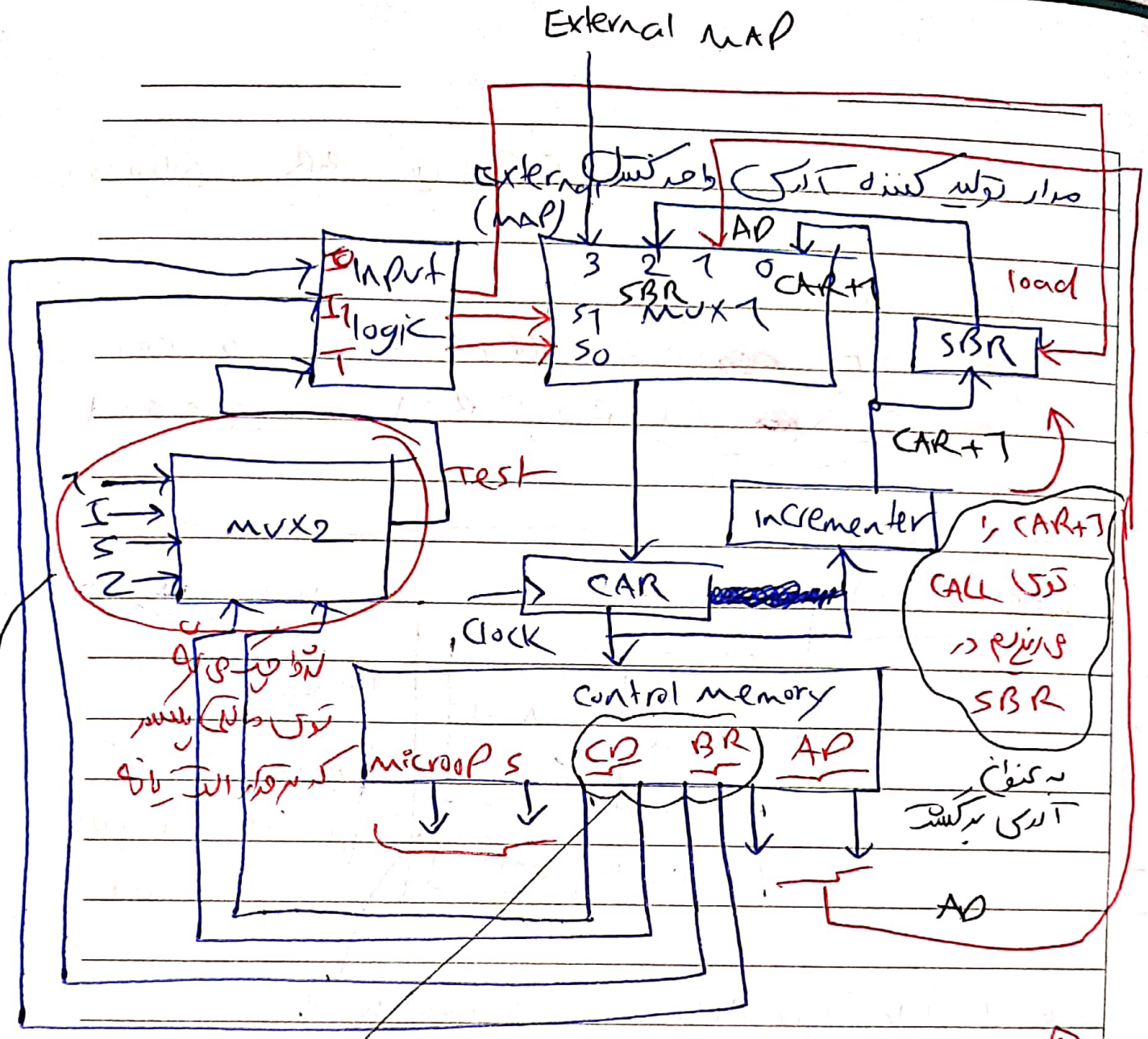
DR را می‌ریزیم توی M[EA] و WRITE و در آخر هم FETCH را فراخوانی می‌کنیم

به کمک دو فیلد CD و BR تعیین می‌کنیم این دستور بعدی از کجا می‌آید

Scanned by CamScanner



Scanned by CamScanner



Condition و Branch →

معلوم می‌کنیم دستور بعدی از کجا می‌آید

چک کردن لودها →

00 01 10 11

U و I و S و Z

اگر 00 بود می‌گیم ادرس Test

یک بود یعنی تنها همواره بتوانه الت

توی این مدار ← مدار [CAR] یک ماکسی ۴ به ۱ می‌گذاریم

چرا؟ چون Jmp و CALL و RET و MAP داریم

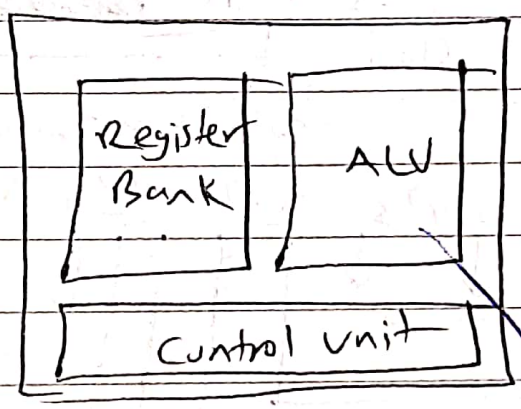
(جدول BR را ببین)

Scanned by CamScanner

برالاکی BR یکی از ۴ حالت را انتخاب می‌کنیم.

BR	T		
۰۰	۰	CAR ← AD	→ خروجی ۰ ۰۰
۰۰	۱	CAR ← CAR+1	→ خروجی ۰ ۰۱

session 78



ساختار پردازنده

اصلی‌ترین قطعه‌ی کامپیوتر که مدیریت سایر بخش‌ها را بر عهده دارد.

واحد محاسبه و منطق

arithmetic logic unit

ALU ۰ داده‌ها را از کجا می‌گیرد؟ از یک سری رجیستر که در قلب پردازنده نوع حافظه در کامپیوتر هستند.

کل Register Banks, ALU ۰ تحت کنترل Control unit هستند.

واحد کنترل

مثلاً ۰ در چه زمانی داده رجیستری ۰ چه داده‌ای از حافظه بیرون ۰ چه پردازشی روی آن انجام شود و به کدام ورودی و خروجی برای ارتباط با کاربر ۰ انتقال داده شود.

Scanned by CamScanner

انواع ساختار پردازنده

جدید قابل ذکر ← رجیسترهای بیشتر با قیمت کمتر در واحد سطح پیشان

① یک ثبات خاص منظوره (مثل اکومولاتور) این طرح خیلی کارا و ارزان هستند ← اما با کلی محدودیت های برای برنامه نویس و اونجایی که ارزان و کارا می شود بوی هم الت

② چند پردازنده های منظوره ← جایی که بوی هم الت و بتوانیم دیتاهای بیشتری را در هم زمان در CPU پردازش کنیم

③ حافظه پشته ④ حافظه تک کش (همین الت های ساختارهای جدید را در خودش داشته باشد)

ذات عملیات با پشته قابل پیاده سازی الت (مثل return کردن توابع)

«ثبات ها»

① دسترسی به آنها در مقایسه با حافظه بسیار سریع تر الت

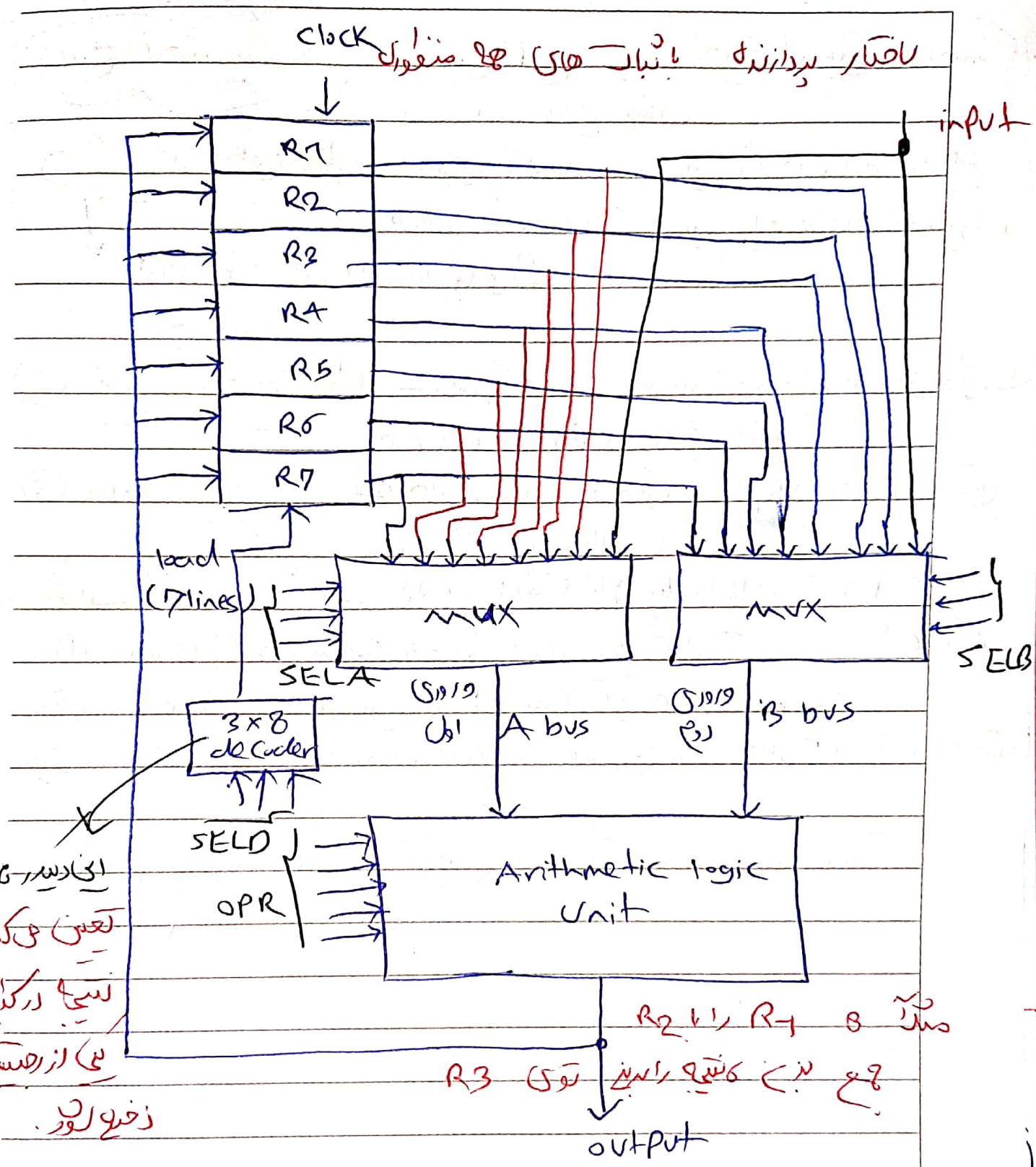
② در کامپیوترهایی تنها یک ثبات (AC) برای عملیات وجود داشت

③ در کامپیوترهای امروزی تعداد ثبات بیشتری برای عملیات در دسترس هستند

دیتاهایی که می خواهیم پردازش کنیم ← حتماً باید بیان توی رجیسترها تا بتوانیم پردازشش کنیم.

Scanned by CamScanner

ساختار پردازنده با ثبات های چهار منظوره



این دیکودر تعیین می کند نتیجه در کدام یک از رجیستر ها ذخیره شود.

مثلاً R1 و R2 را جمع بزن و نتیجه را بریز توی R3

تفاوت با کامپیوتر پایه: ورودی و خروجی ALU که هر یک از این ۷ رجیستر می تواند باشد. ۸ حالت → از این ۷ رجیستر و input

★ توی این کامپیوتر ← باید مبدأ اول و مبدأ دوم و مقصد را تعیین کنیم (البته چه کاری هم انجام شود)

(مهم است)

ساختار دستور

SELA	SELB	SELD	OPR
3	3	3	5
3 بیت	3 بیت	بیت	بیت

OPR select	operation	symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	Add A+B	ADD
00101	Subtract A-B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	shift right A	SHRA
11000	Shift left A	SHLA

این برنامه خیلی بهینه نیست

مثال: $(op1+op2)\times(op3+op4)$

① برای AC: LDA op1

② برای چند منظوره:

```
LDA op1
ADD op2
STA SUM
LDA op3
ADD op4
MUL SUM
```

این دستورات مدام با حافظه (بهینه نیست) کار دارند

```
LDA R1, op1
LDA R2, op2
LDA R3, op3
LDA R4, op4
ADD R5, R1, R2
ADD R6, R3, R4
MUL R7, R5, R6
```

سه تا از دستورات اولی هیچ کاری با حافظه ندارد

اگر بنویسیم $(op1+op2)\times(op1+op2)$

① AC →

```
LDA op1
ADD op2
STA SUM
LDA op1
ADD op2
MUL SUM
```

② چند منظوره →

```
LDA R1, op1
LDA R2, op2
ADD R5, R1, R2
MUL R7, R5, R5
```

از نظر کد سرعتی ضمانت کلوک

جمع کد کمتر

حافظه کمتر

R7 برنج توی

R5, R5 را در هم ضرب کن

مزیت ها ← ① کمتر با عملوندی که و کد داریم

② (از تعداد عملیات ← کد ← حافظه)

---

main

...

call func1

a1 8

func1

call func2

a2 8

func2

...

Stack

call func3

a3 8

return

در آخر func1

a1, a2, a3

اول این را return می کنیم

بعد فانکشن ۲ رسیدن کلوک

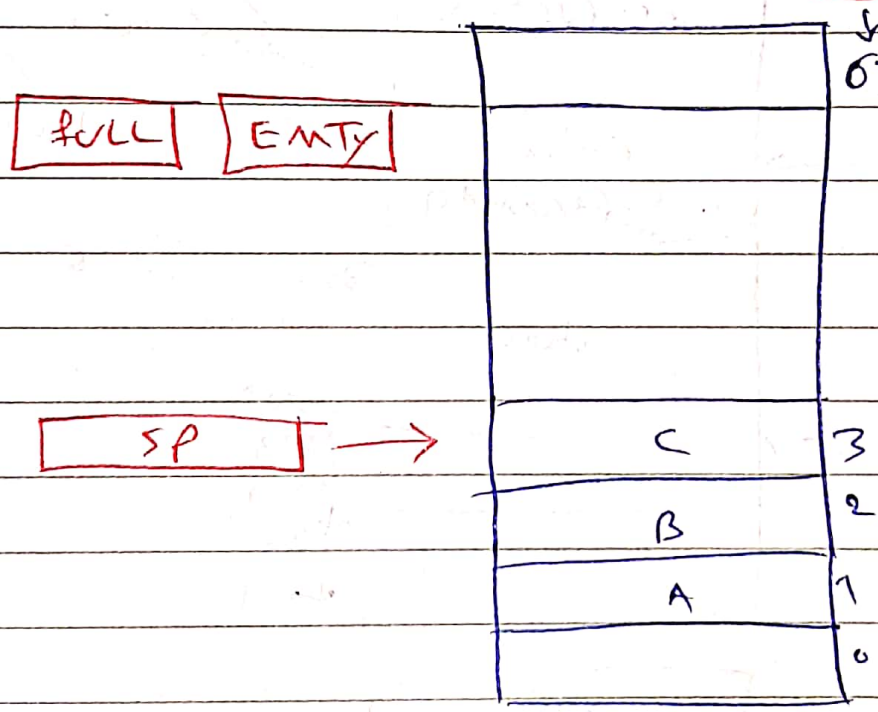
پس موقع فراخوانی اول a_1 و a_2 و a_3 اما موقع return
اول a_3 بعد a_2 و بعد a_1 را بر می‌داریم

پس در فراخوانی تابع آدرس برگشت را در الگ ذخیره می‌کنیم

* برای کار با پشته یک ثبات $\boxed{SP}$ ← stack pointer التفاف می‌شود

* حافظه پشته می‌تواند به صورت ثبات یا درون حافظه اصلی پیاده شود.

حافظه به صورت ثبات



یعنی سرفاکی نیست الگ

نوشتن در پشته

تحت عنوان عمل PUSH کردن نامیده می‌شود.

① $SP \leftarrow SP+1$ (stack pointer افزایش می‌شود)

② $M[SP] \leftarrow DR$ ، $Empty \leftarrow 0$ ← اولین متغیری که پشته می‌کنیم

③ $SP = 0$ ؟ $FULL \leftarrow 1$

در آدرس ⑦ ذخیره می‌شود

اگر $SP = 0$ ؟ → $\boxed{FULL = 1}$
الگ پر شده

دیگه الکی پر نشه (0 می‌ذاریم توی FULL)

خواندن از پشته ← (pop)

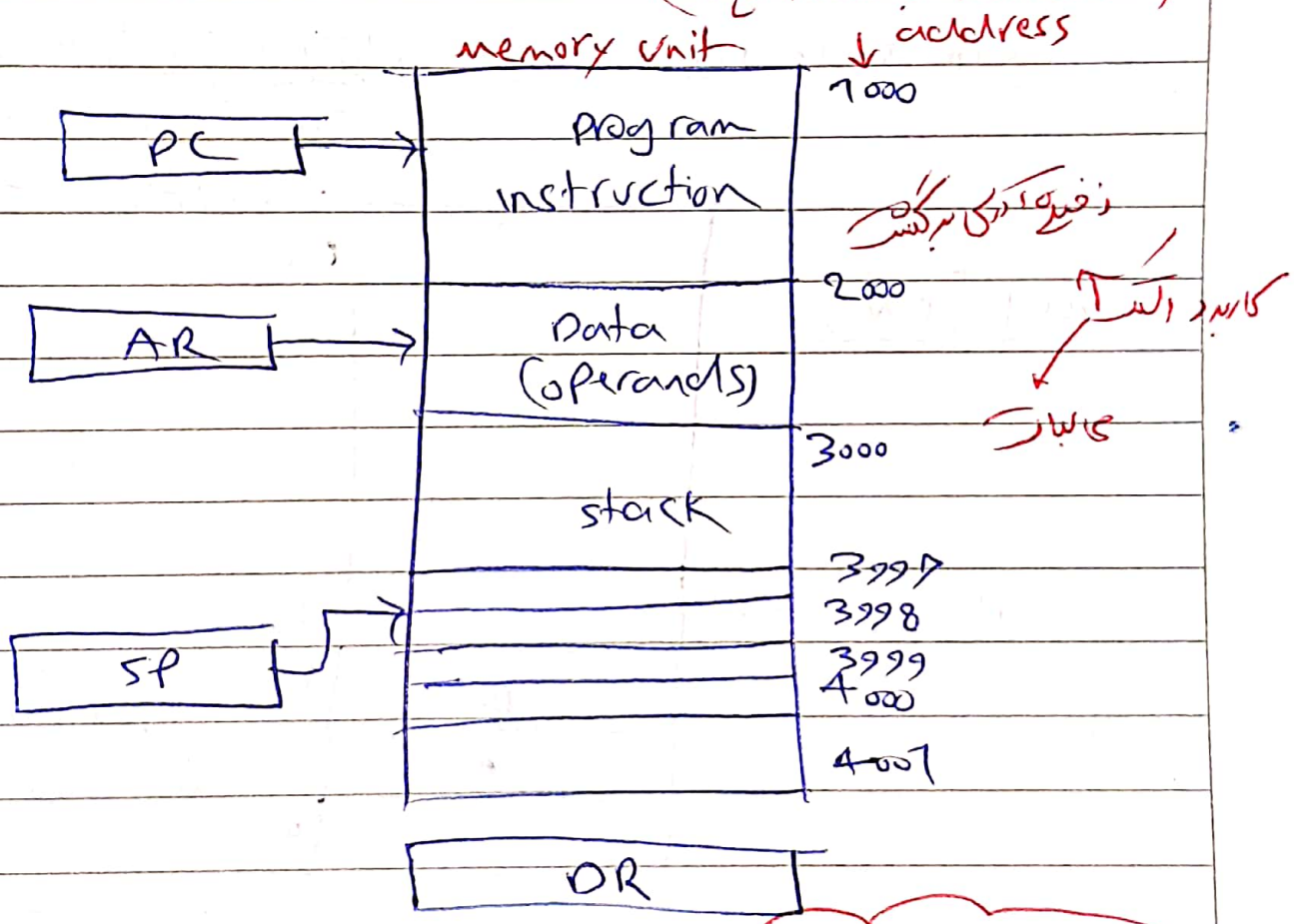
① DR ← M[SP]

② SP ← SP-1 , FULL ← 0

③ SP = 0 ; Empty ← 1

یعنی الکی خالی → اگر SP=0 بود

(ساختار حافظه با حافظه پشته)



قاعده معکوس لهستانی

(3*4) + (5*6)

اول operand ها را قرار بدهد بعد اپراتورها را

معادل لهستانی B. $\boxed{34*56*+}$ → عملیات به کمک پشته

بازه کردن عملیات در پردازنده

صفوی

هرجا که متغیر دیدیم push کنیم تو استک، به operator که رسیدیم 2 متغیر را pop کنیم به جای کمش 1 نتیجه را ذخیره کنیم

34 *56 * +	push 3
	push 4
به جای اینکه از register AC ها برای	pop 3, 4
انتقال داده به ALU استفاده شود	push 3×4
از استک استفاده می‌شود	push 5
خانه‌های آخر استک به ALU وصل است	push 6
	pop 5, 6
	push 5*6
	pop 12, 30
	push 42

push op1
push op2
MUL
push op3
push op4
MUL
ADD

برنامه‌ی CPU به کمک استک

نتیجه را می‌ذاره تو op3

برنامه‌ی CPU → به جای register

session 19

«انواع پردازنده برحسب دستورات» instruction set, ISA, Architecture

«حجم بیشتری در حافظه می‌گیرد»

① RISC — Reduced Instruction set computer
- دستورات کوتاه و ساده ← بهتر کار می‌کنند اپریشن‌ها
- طول دستورات ثابت
- تعداد دستورات بیشتر برای یک برنامه
- وقتی دستورات ساده‌اند کدام بخش CPU ساده می‌شود؟ control unit

② CISC — complex Instruction set computer
- دستورات پیچیده و طولانی
- دستورات با طول متغیر
- تعداد دستورات کمتر برای یک برنامه

دیدگاه روی سعی کنیم مسائل را در سخت افزار حل کنیم چون کمتر افزار
لازم تر از نرم افزار است ← $\boxed{\text{کد}}$

طول دستورات بنا به نوع دستور عوض می‌شه. (مثلاً از ۴ بایت تا
۳۲ یا ۶۴ بایت)

مشکل این ساختار ما چون پیچیده است ← گران تر تمام
↑ زمان طراحی و تست

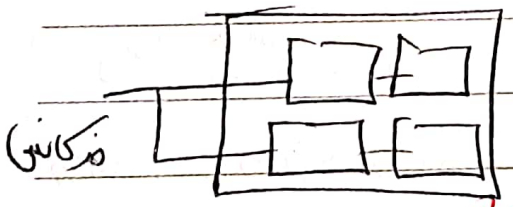
Performance
بستگی به بارگی و یا پیچیدگی
سخت افزار ما دارد.

اما طبق قانون Moore کیفیت ایجاد
زیاد اهمیتی ندارد روی تاخیر کدن

↑ پیچیدگی سخت افزار ← $\boxed{\text{↑ تاخیرها}}$

* اگر کلاک را متفاوت بسازیم (Async باشه) ← طراحی خیلی سخت می‌شود
آسنکرون

اگر کلاک Sync باشه ←
فرکانس متناسب با کندترین واحد این سیستم می‌شود



همه واحدهای سیستم را پیچیده می‌کنیم

کندترین واحد تا کندتر و کندتر می‌شود ←
روی همه‌ی کنش‌های سیستم تاثیر منفی می‌گذارد.

طبق این دلایل

$\boxed{\text{CISC}}$ کنار گذاشته شد و برای پردازنده‌های با performance بالا
RISC را دارج پیشرفت می‌دهند

مثلاً RISC 100 تا دستور دارد

و CISC 5 تا دستور

اما چون RISC کلاک بالاتری دارد ← مثلاً هر دستور 3ns

و CISC هر دستور را 100ns

100 تا دستور	3ns	
5 تا دستور	100ns	

→ RISC ✓

بدرد تر

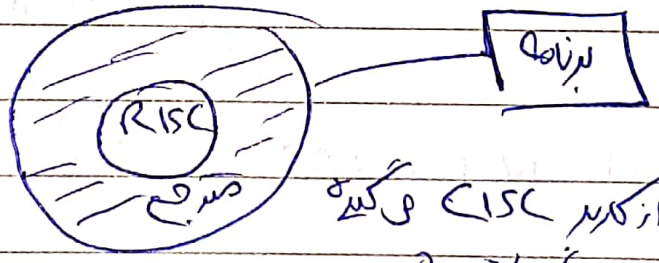
چون فرکانس متناسب با ولتاژ تری دستور تقسیم می‌شود

کلی نیستم

چون انعطاف زیادی به برنامه‌نویس

تنها جایی که CISC هنوز وجود دارد ← پردازنده‌های Intel است

Intel ← هسته‌ی پردازنده‌هایش RISC اما پوسته‌ی CISC دارد



برنامه را از کاربر CISC می‌گیرد

تا انعطاف زیادی داشته باشد اما CISC

لایه‌ی میانی که یک مترجم دارد ← دستورات پیچیده را به بازده تبدیل می‌کند و به RISC می‌دهد

Core I7 ← RISC5 را دارد

R1, R2 را جمع کن و بریز توی R3

★ انواع دستورات بر اساس تعداد عملوند ★ توی RISC

① سه عملوندی

ADD R3, R1, R2

② دو عملوندی

MOV R1, ADDR

MUL R1, R2

دو عملوند هم هست

R1, R2 را در هم ضرب کن نتیجه را بریز توی عملوند اول (R1)

Scanned by CamScanner

ADD OPR1 ③ تک عملوندی

④ صفر عملوندی ~~PUSH A~~ POP
~~PUSH B~~

انواع دستورات برای عملکرد ← AND ، OR ، XOR ، ADD

① انتقال داده ← RCR ، ROR ، SHR ، SHL

② محاسبات و منطق و شیفت ← ADD ، SUB ، MVL ، DEV

③ کنترل روند برنامه و شرطی و غیر شرطی

اول بحث کار در لیست داده ها که قرار گرفته اند توی حافظه
که می توانیم این داده ها را پردازش کنیم و توی رجیسترهایی که روی پردازنده
هستند ←

حداقل چیزی که نیاز داریم این است که بین رجیستر و حافظه بتوانیم
دیتا رو نقل و انتقال دهیم.

حداقل ها ← شیفت و ADD و NOT باید باشند تا بتوانیم چیزهای
کلی را بنویسیم.

روش های آدرس دهی

① آدرس دهی فوری توی خود عملوند operand دستور مشخص هست قرارش روی

② آدرس دهی بلافصل جای دیگری دستور انجام شود ← CMA
متمم

آیا می توانیم بگیم فوری مساوی متد عملوندی است؟!
نه لزوما

آدرس شیفت

ADD 5 ← AC+5
AC

← Immediate

تک عملوند
خود عملوند operand

ADD 5 ⟵ 5 ← آدرس دهی بلافصل (مستقیم)

AC ← آدرس دهی ضمنی الد

③ آدرس دهی ثبات

ADD R1, R2

R1, R2 را جمع کن بریز توی R1

* توی کامپیوتر باید نداشتیم این نوع را

④ آدرس دهی غیر مستقیم با ثبات

ADD M[AR]

آدرس را بردار توی AR

اول برو AR را بخوان

بعد میایم M[AR] را می‌خوانیم

دیتا و آدرس‌ها توی CPU و در یکی از register های

نیازی نیست برم توی حافظه آدرس را بیارم ⟵

(با ثبات کار می‌کنیم)

⑤ آدرس دهی مستقیم

ADD X

ADD M[500]

⑥ آدرس دهی غیر مستقیم

ADD X I

حالت‌ها

ADD R1, M[R2] ← چون آدرس را توی رجیستر R2 نوشتیم

ADD R1, M[500] ← غیر مستقیم با ثبات

ADD R1, M[500] I ← مستقیم

ADD R1, M[R2] I ← آدرس دهی غیر مستقیم

غیر مستقیم

⑦ آدرس دهی افزایشی (کاهشی) خودکار

ISZ X

پیاده‌سازی کامپیوتر

یک واحد افزایش

Scanned by CamScanner

توی Basic نبود

⑧ آدرس دهی نسبی relative

BUNR X

branch unconditionall relative 50

وقتی این دستور fetch میشه ← PC میره 151

فرض کن ما توی خانه‌ی 100 هستیم

100 BUNR 50 — BUNR میگه نسبت به آنجایی که الان هستیم (101) 50 تا بره جلو (نسبی عمل می‌کنه) میگه می‌تونستم بگم BUNR _50

101 ← 151 میاد اینجا

از 101 میره خانه‌ی 51 ← BUNR _50

خاصیتی چیه؟ (پیاده‌سازی راحت‌تر حلقه‌ها)

⑨ آدرس دهی پایه و اندیکس

MOV [BP][CX] — BP + CX

BP: Base pointer — نسبت به آدرس پایه‌ی BP به اندازه‌ی CX بره جلو

MOV [100][5] → خانه‌ی 100

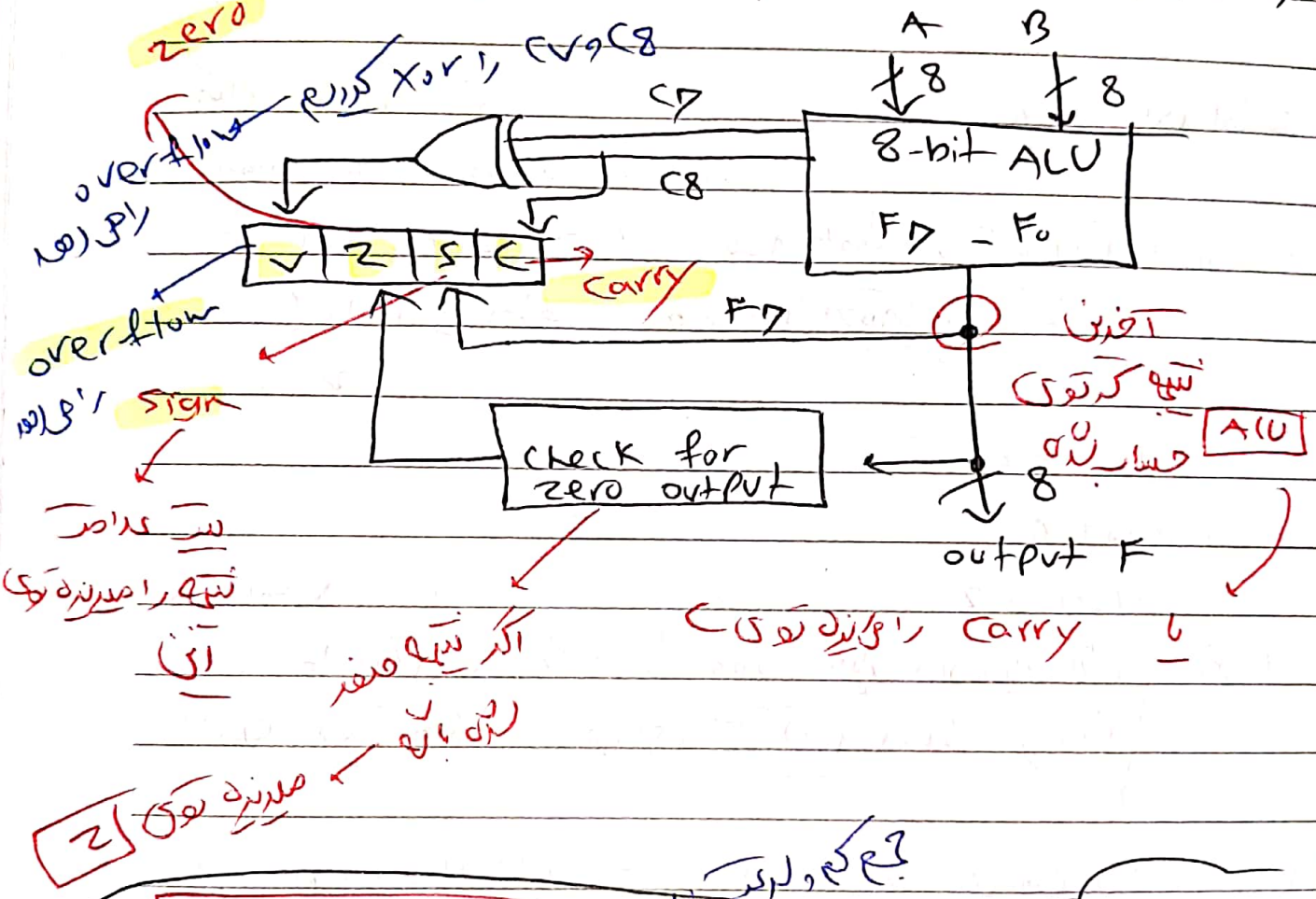
MOV [100][1] → خانه‌ی 101

برای کار با دردی خوره آرایه‌ها

یک رجیستر جدا که کلی رانه می‌دارد

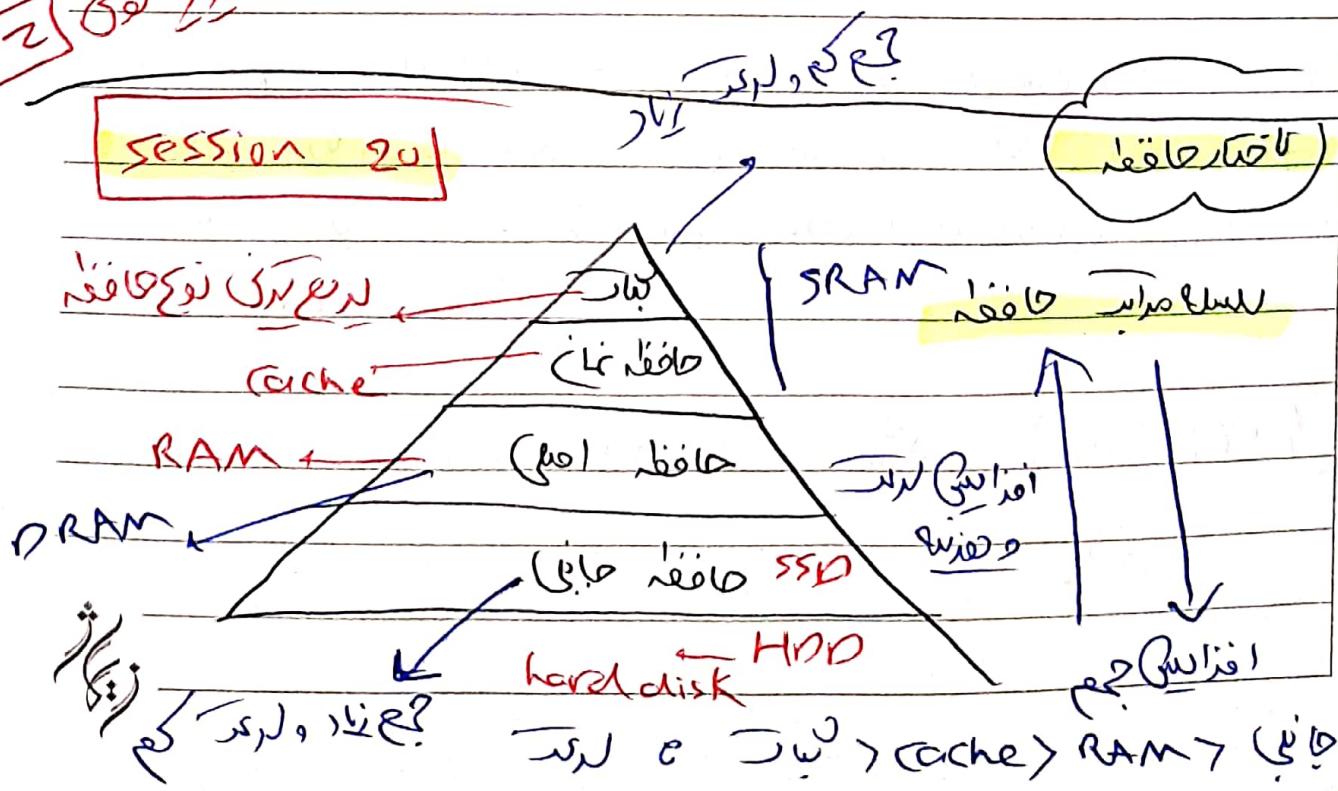
Program status word [PSW] کلمه وضعیت برنامه

* بیت های پرچم که نشان دهنده وضعیت نتیجه اجرای دستورات هستند.
* در دستورات کنترل روند برنامه به آن ها رجوع می‌شود.



Session 20

ناحیه حافظه



Scanned by CamScanner

خاصیت محلی بودن

زمانی که داده‌ای که الان در حال پردازش است به زودی نیز مورد پردازش قرار می‌گیرد

مکانی که داده‌ای و داده‌های اطراف آن نیز مورد پردازش قرار خواهند گرفت.

مثلاً اگر n توی خانه‌ی 100 حافظه است و قاعدتاً دستورات بعدی به داده‌های اطراف n را هم احتیاج پیدا می‌کنیم

700 n

706 y

702 z

* نکته

اگر ما نیازی داریم که فقط وقت بسیار مورد نیاز باشد و دگر نخواهیم لسلسه مراتب حافظه را طولانی‌تر کنیم ← باعث می‌شود

پس وقتی این لسلسه کمک می‌کند که خاصیت محلی بودن را داشته باشیم یعنی یا مدت زمانی به این متغیر نیاز داشته باشیم توی برنامه‌ی یا نیاز کردن ما از اینکه بخواهیم از حافظه‌های پایینی لسلسه که کندتر هستند استفاده کنیم

هر چقدر بتوانیم کپ کارایی حافظه را بیشتر کنیم ← سرعت CPU ↑

پیدا می‌کند

کاربرد خاصیت locality مکانی چیه؟

مثلاً می‌گه تو اگر n را توی خانه‌ی 100 خواندی احتمالا 104 تا کلمه‌ی بعدیش را هم کش داری

یعنی مثلاً

1000	n
1001	x
1002	z
1003	.
⋮	⋮

اگر خواستیم n را بخوانیم بدون این خاصیت

تا خیر اولیه‌ی فعال کردن آدرس

104

مثلاً ما دیگه n فقط 104 زمان میبره

اجرایش 104 ←

در کل ما $\boxed{144}$ ← 4 تا دستور ← $4 \times 104 =$

مثلاً داریم $\boxed{443}$

$\boxed{4 \times (104 + 4)}$

اما اگر بیایم در حین خواندن آدرس n ما آدرس 4 تا دستور دیگر را هم بخوانیم ← یعنی $\boxed{104}$ برای خواندن و $\boxed{43}$ برای اجرا

$\boxed{143}$ می‌شه ← در مقایسه با 443 خیلی بهتر است

$\boxed{104 + 43}$ در موقع خواندن از حافظه ← به جای کلمه کلمه خواندن، بلوک بلوک می‌خوانیم

به خاطر تاخیر اولیه آدرس ← مثلاً 4 تا 4 تا می‌خوانیم

حافظه اصلی

① الگاید ← رفتاری ما خیلی خوب که با 4 تا ترانزیستور پیاده‌سازی شده

دائمی است

سرعت زیاد

حجم زیاد کمتر افزار روی تراشه (خیلی) ← گرانی قیمت

زیاد

الدایدع

Scanned by CamScanner

لرزد خازن کمتر از ترانزیستور الت و همچنین باید refresh کنیم ← لرزد کندتر

② دینامیک ← یک سلول با یک ترانزیستور و یک خازن پیاده سازی می‌شود
RAM

جمع کمتر سمت افزار ← ارزان‌تر

نیاز به تازه کردن (شارژ مجدد)

خازن یک جریان نشتی دارد ← اگر بیت ورودی تخلیه می‌شود ← ندارد

یک مداری می‌خواهد که داده‌های بیت‌های حافظه را بررسی کند و پیشنهادهایی که بیت یک دارند را دوباره به "یک" تولید بنویسد تا اگر ولتاژ "یک" دوره پایین بره بالا ←

refresh کردن حافظه ← تازه‌سازی ← زیاد لرزد

* حافظه‌های cache های نزدیک به CPU ← با دینامیک پیاده سازی می‌شوند ← register

پایین‌تر ← از دینامیک استفاده می‌کنند

DDR با حجم زیاد ← Dynamic RAM (۳۲ گیگ)

SRAM ← static RAM دیگر لازم نداریم چون گران است

انواع دسترسی به حافظه (اصلی) (بیشتر جاها وقتی می‌گیم حافظه چیه منظورمون حافظه اصلی است)

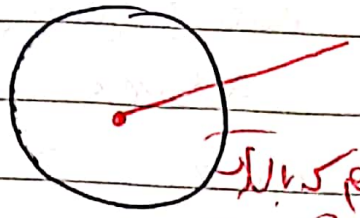
(RAM) دسترسی تصادفی ← RWM ← read & write memory

ROM ← read memory

نوشتن + خواندن

فقط خواندنی

دیتای تصادفی: برای رسیدن به یک نقطه خاص، باید تمام دیتاهای قبلی را پشت سر بگذاریم

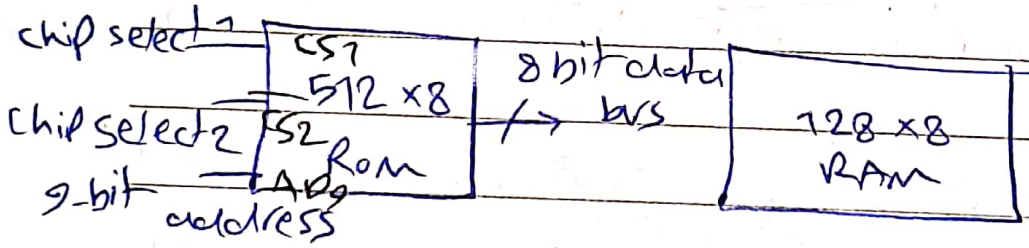


یک دیسک داریم که بالای آن خاص در حال گردش هست

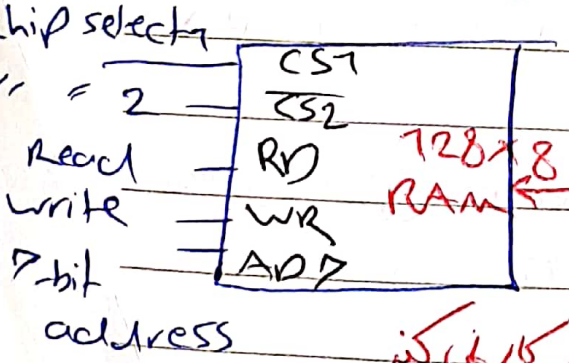
فضای حافظه‌ی CPU → (2^n - 1 تا صفر) «ظرفیت آدرس حافظه»

یک سیستم با کالت بایت حافظه RAM و کالت بایت حافظه ROM

تراشه‌های پایه حافظه: RAM های 128 تایی و ROM های کالت تایی



512 = ROM و 128 = RAM که



تا وقتی chip select فعال نباشد ← حافظه کار نمی‌کند

★ توی این مثال ← CPU قرار است که 1024 خانه‌ی حافظه را آدرس دهی کند پس حداقل 10 خط آدرس می‌خواهد

پس CPU از 0 تا 1023 آدرس دارد.

Scanned by CamScanner

این 10 خط آدرس باید به چه‌جوری به آدرس‌های 7 تایی و 9 تایی وصل شود ←

یک جدول تشکیل می‌دهیم.

Component	Hex address	Address bus 10	9	8	7	6	5	4	3	2	1
RAM1	0000 - 007F (صفر - 127)	0	0	0	x	x	x	x	x	x	x
RAM2	0080 - 00FF	0	0	1	x	x	x	x	x	x	x
RAM3	0100 - 017F	0	1	0	x	x	x	x	x	x	x
RAM4	0180 - 01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200 - 03FF	1	x	x	x	x	x	x	x	x	x

ROM ← 512 تایی ← $\boxed{\text{صفر آدرس}}$ 128 خونه

پس 9 بیت کم‌ارزش را به آدرس ROM وصل می‌کنیم

chip select ← همیشه بیت دهم = یک باشد ← یعنی اگر بیت دهم یک باشد فقط ROM فعال می‌شه

یعنی امکان نداره که CPU یک آدرسی که قرار می‌دهد که دوتا از این حافظه‌ها فعال شود. یا RAM1 فعال می‌شه یا RAM2 یا 3 یا 4 یا ROM دهم دوتایی هم‌زمان فعال نمی‌شوند

معادل آدرس بالای همیشه خانه‌ی صفر RAM3

معادل آدرس 512 تبدیل خانه‌ی صفر ROM

CPU متوجه این دیتاها نیست ← $\boxed{\text{به این کاری کن تا لیست آدرس حافظه}}$

یک سیستم با 128 بایت حافظه RAM و 512 بایت ROM

جدول :

component	Hexaddress	address BUS
RAM	0000-007F	000XXXXXXX
ROM	0080_00FF	001XXXXXXX
	0100_017F	010XXXXXXX
	0180_01FF	011XXXXXXX
	0200_027F	100XXXXXXX

* قانون ها ① کارکرد درون تراشه باید از تمام صفر شروع شود و به تمام یک ختم شود

② کارکرد بیرون تراشه باید از ابتدا تا انتها کامل باشد

توی این جدول این دو قاعده رعایت نمی شود برای ROM و از تمام صفر به تمام یک نمی رسیم و همچنین از بیت های پرارزش هم نمیشه تریت را تشخیص داد

چون 128 تای اولی (برای RAM) تریت های ROM را 128 تا تحت دارد

* برای فعال کردن ROM با ۳ بیت چون هیچ قاعده ای بنشون نیست مشغول نمی توانیم MAP را انجام دهیم

001 و 010 و 011 و 100

تریت مشغول ندارد

راه حل: اول ROM را MAP می کنیم و بعد RAM را

جای RAM و ROM را عوض می کنیم

ROM	ROM	0000_017F	00XXXXXXXX
	RAM	0200_027F	010XXXXXXX

می دهیم ROM

زیرا حافظه دارد

ROM وقتی فعال می شه که بیت دهم صفر باشد و ۹ بیت کم ارزش را به

برای RAM هم از ۷ تا صفر شروع می‌شه تا ۷ تا یک و
کیدیش هم زمانی الت که بیت‌های پرارزش $\boxed{010}$ داشته باشد

در نتایجت آدرس حافظه ←

ابتدا حافظه با حجم بیشتر را نشان می‌کنیم

$\boxed{\text{Session 21}}$

حافظه‌ی Associative

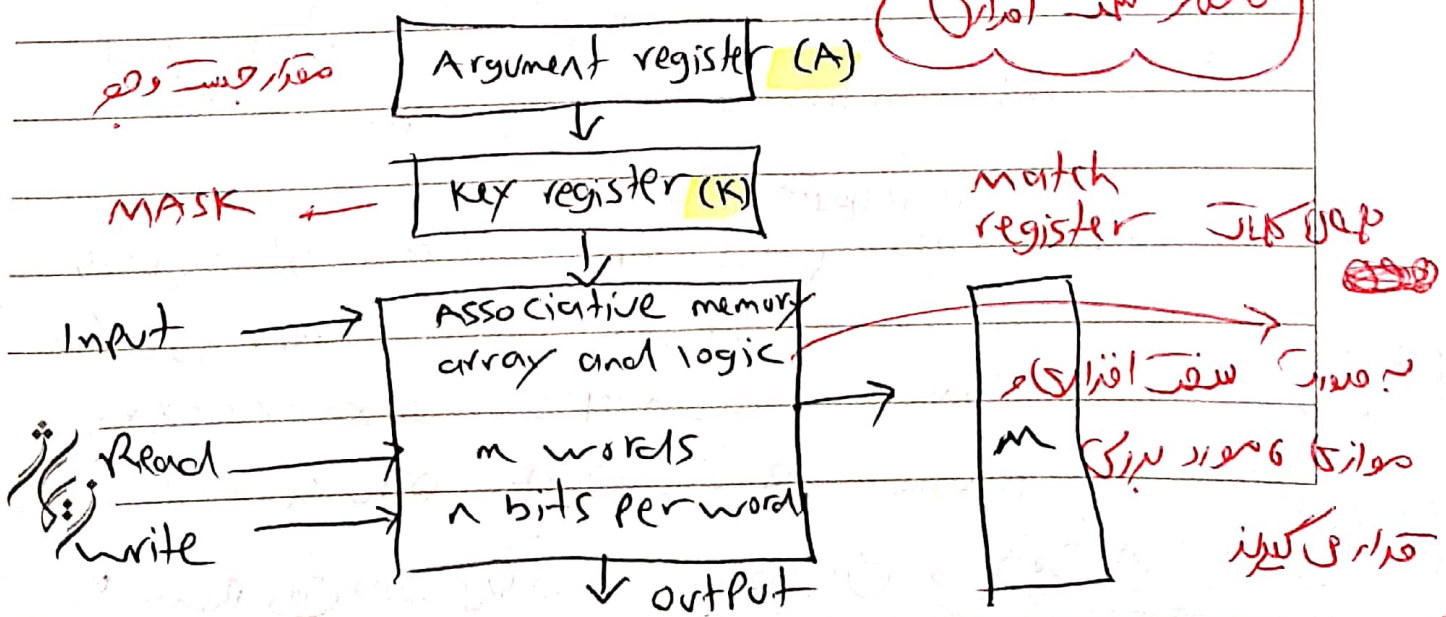
شرکت پذیر

ذخیره و استخراج اطلاعات یک جدول

* پایگاه داده و دیکشنری و ...

* روش نرم افزاری (پیدا کردن دیتا) تاخیر زیادی دارد
پس به جای نرم افزار، سخت افزار حافظه‌ی Associative را داریم
که با سرعت خیلی بالا در یک کلاک ساعت می‌توانند کلیدهای مقادیر حافظه را
مورد جست‌وجو قرار دهد و دیتای مورد نظر را lookup کنند

ساختار سخت افزاری



مثال: A 8 1 0 1 ← مقدار جست و جو

mask ← K 8 1 0 0 1

در مرحله اول این دو تا را با هم ترکیب می‌کنیم ← می‌گه آنهایی جست و جوی

تو ← 8 ← P 8 1 X X 1

pattern

بیت دوم و سوم ← mask = صفر ← توی جست و جو

مقدار این دو تا بیت مهم نیست چی باشن ← [X] می‌گذاریم

dont care

و بیت ← چهارم ← mask = 1 ← جاهایی که توی A

هست، جست و جو می‌شود ← این جوری pattern را تشخیص داریم.

پس هر جایی که بیت اول و آخرش یک باشد را مورد جست و جو قرار بده

هر کلمه‌ای که این تطبیق را داشته باشد ← بیت متناظرش در

register ← M، یک می‌شود و هر کلمه‌ای که تطبیق نداشته باشد ← بیت

متناظرش، صفر می‌شود

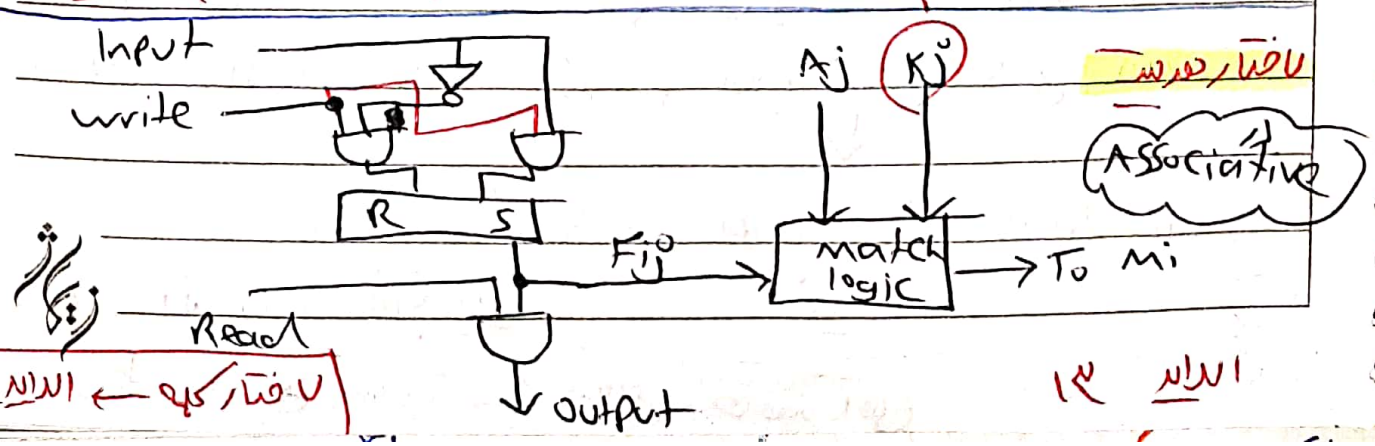
* جهان کلمات به صورت تنها افرادی و موازی مورد بررسی قرار می‌گیرند

پس توی یک کلاک تاکت، جهان اون کلمه‌هایی که تطبیق پیدا کردند

بیت متناظرشون یک می‌شود ← می‌توانیم در زمان پردازش را دوباره

انجام دهیم.

بیت را توی جستجو، مورد پردازش قرار دهیم یا خیر؟



ساختار دوست

Associative

زیگزاگ

ساختار کلمه ← اندازه ۱۴

اندازه ۱۲

وقتی تاخیر کمتر بیشتر دارد.

سرعت کلاک این حافظه کمتر از سرعت کلاک حافظه معمولی است

و همچنین باعث گران تر شدن و پیچیده تر شدن می شود

حافظه نهان — Cache — RAM

از نظر هزینه ← ارزان تر از ثبات و گران تر از اصلی

از نظر حجم ← از ثبات بزرگ تر و از اصلی کوچک تر

طبق خاصیت locality ← کاهش اختلاف سرعت رجیسترها و اصلی

کم کردن کپ

بخشی از دیتای که بیشتر باهاش کار داریم از دیتای اصلی میره تو cache و شروع به خواندن دستورات می کند از cache

در موقع که رجیسترها دیتای جدیدی بخواهند ← اول توی cache دنبالش می گردند

اگر توی cache بود ← می آورد

اگر نبود میره توی حافظه ی اصلی و از این

میره توی cache و تحویل register می دهد

پس

① ابتدا آدرس در حافظه نهان بررسی می شود که آیا وجود دارد یا نه

② اگر وجود داشته باشد ← اصابت (hit) رخ می دهد

نیازی به دسترسی به حافظه اصلی نیست

③ اگر وجود نداشته باشد

عدم اصابت ← ارجاع به حافظه اصلی

Scanned by CamScanner

تعداد hit

hit ratio = hit / access ← ترکیب ② و ③

access = hit + miss

هدف بهبود حافظه‌های پایین‌تر بدهیم

کشف می‌شود

نرخ اصابت = تعداد اصابت / تعداد کل دسترسی‌ها

اگر hit ratio کم باشد ← مجبوریم تا خیلی cache را تحمل کنیم و با احتمال کمی نیاز را توی cache پیدا می‌کنیم و با احتمال زیادی پیدا نمی‌کنیم و باید بدهیم دنبال حافظه اصلی

پس وقتی hit ratio کم باشد ← یعنی داریم با حافظه اصلی کار می‌کنیم و فقط الکی یک تاخیر cache هم اضافه کردیم ← ضرر

پس باید طوری باشد که نرخ این اصابت حداکثر بشه ← هدف بدیم

نزدیک تر شود ← بهتر می‌شود

نرخ اصابت به ← خالی نیستم بسته دارد.

باید cache

اما باید cache هم عمر آلت

افتاد برنامه

کاری ← اصابت بیشتر می‌شود ← هدف علی لوین بیشتر باشد

نیستم

دلایل عدم اصابت

مدل 3Cs

cold ← اول کار خالی است cache

conflict

capacity

زیاد

RAM و cache

cache مثل RAM ← وقتی نیستم روشن می‌شود و نیاز پیدا بدهد توی cache

Scanned by CamScanner

پس قطعاً اول کار دفعه کدام اصابت می‌کنیم تا به صورت دیتاها را از حافظه اصلی لود کنیم بیاره توی cache

(البته این داتاها رو توی RAM هم داریم ؟)

از هارد دیسک ← توی RAM ← cache ← در آخر دیتا CPU و احضار می‌شود

capacity ← محدودیت حجم cache

دیتا جدیدها که می‌خواهند وارد بشوند ← قدیمی‌ها را بیرون می‌اندازیم

اگر قدیمی‌ها را دوباره بخواهیم چی؟!

زمان دسترسی به حافظه

conflict ← به

زمان دسترسی RAM × (نرخ اصابت − 1) + زمان دسترسی به cache = زمان دسترسی CPU به دیتای مورد نظر

اگر اصابت داشتیم ← زمان دسترسی مساوی = زمان دسترسی به cache می‌شه

اما اگر اصابت نداشتیم ← حافظه اصلی هم حساب می‌شود

موارد مهم در رابطه با حافظه نهان

جای دهی اطلاعات ← بلوکی که از RAM خوانده شد را کجای cache بگذاریم؟

سیاست‌ها ← تشخیص وجود اطلاعات ← چطوری دیتا را توی cache پیدا کنیم؟

جایگزینی اطلاعات ← cache اگر پر شد ← دیتای جدید چطوری بیاید توی cache؟ ← کدام بلوک را بندازیم بیرون و با کدام جایگزین کنیم؟

به روزرسانی اطلاعات

دیتاها را آوردیم توی cache و پردازش کردیم ← اگر مقدار جدید گرفتن ← چه موقع توی RAM هم

آدرس دهی مستقیم

سیاست های جای دهی اطلاعات ← آدرس دهی تماماً شرکت پذیر

آدرس دهی شرکت پذیر مجموعه ای

① آدرس دهی مستقیم direct map ←

حافظه نهان از بلوک هایی هم اندازه با حافظه اصلی تشکیل شده

(دیتاها را نگه می دارد.)

valid = 1 و Tag (آدرس بالا)

valid	Tag	Block 0
valid	Tag	. . . .
valid	Tag	Block N-1

تعداد بلوک در cache % آدرس بلوک در حافظه (اصلی) = شماره بلوک در حافظه cache

باقی مانده

مشخص کردن آدرس بلوک در cache

Tag	Block No.	word offset

آدرس در حافظه اصلی

آدرس بلوک در حافظه اصلی

مثلاً ۲ بیت بالا ← آدرس بلوک مورد نظر توی cache

یعنی اگر صفر شد ← توی Block 0 cache

یک شد ← 1

دو شد ← 2

سه شد ← 3

حافظه cache

* هم زمان نمی توانیم آدرس صفر حافظه و ۶۴ حافظه را بگذاریم توی آیدیت کنیم

چون هر دو تا توش به یک بلوک در cache و map می‌خورند

مثال: CPU در دستورات اولش با آدرس 5 حافظه کار داشته باشد
و توی دستورات بعدی با آدرس 69 کار داشته باشد

فرض: هر بلوک 16 کلمه و cache 4 تا بلوک داره

باقی مانده آدرس بلوک بر تعداد بلوک

Address 5: 00005 → 4 بیت کم ارزش = آدرس کلمه در بلوک

هفت

Address 70: ... 00046

هفت

4 = باقی مانده آدرس بر تعداد بلوک = 4 → 4 % 4 = 0

صفر

صفر

هر دو تا توش به یک بلوک در cache و map می‌خورند
دلیل دومی که می‌بریم ← اولی را می‌اندازه بیرون با اینکه توی cache هنوز 3 تا بلوک جا داریم.

اگر موقع پیاده سازی قبلی کار داشته باشیم، می‌گیم conflict رخ داده

هر دو تا با اینکه جا داریم
توی یک نقطه می‌خواهند قرار بگیرند
در cache

فرق بین پیاده سازی پر و خالی را چطوری بفهمیم؟

بیت valid ← اول کار صفر است برای همه‌ی بلوک‌ها
اولین بلوکی که مقدار توش قرار گرفت ← بیت valid را یک می‌کند

زیبا

* دیتای دوم (V0) را که می‌خواهیم بخوانیم، valid که یک بوده
خوبی نفهمیم که همان بلوک مورد نظرما توی بلوک اولی است
یا یک بلوک دیگری توی هست؟
BLOCK

راه: از اون باقی مانده‌ی آدرس را نگه می‌داریم تا از اونی بفهمیم
کدوم بلوک توی cache اومده
۳ بیت - هشت

Tag برای بلوک اول: 000 (00) → ۴×۴ + ۲ = ۱۴ بیت
۲ بیت
که می‌شود

Tag بلوک دوم: 000 (01)

پس در آخر خوبی نفهمیم بلوک توی cache هست؟
اگر تگ توی حافظه‌ی cache، با تگ مورد نظرما یکسان بود
اگر نبود: برای حافظه اصلی
می‌رویم

می‌گیم بلوک توی cache هست

این conflict که نم داره باعث می‌شود، hit ratio پایین بیاد
خیلی فرق طبق اصل locality، با آدرس 5
هم‌چنان کار داریم ولی ما انداختیمش بیرون.

راه حل: به جای حافظه‌ی معمولی برای cache از حافظه‌ی Associative
استفاده می‌کنیم.

Scanned by CamScanner

آدرس اصلی کاملاً لرکت پذیر

← به کمک حافظه لرکت پذیر

در بلوک حافظه اصلی بدون محدودیت
می‌تواند در هر بلوک از حافظه نهان قرار بگیرد

Tag	word offset

جستجوی پیدا کنیم ← باید کل حافظه را اسکن کنیم که ببینیم آیا بلوکی با تگ مورد نظر ما پیدا می‌شود یا نه ←

این اسکن باید با سرعت خیلی زیاد انجام بشه ← التقاری اش Associative

محدوده

مشکل این راه چیه؟ گران اش ←

اگر بخاهیم یک حافظه که حجم زیاد داره را کاملاً Associative پیاده سازی کنیم ← گران و کند می‌شود

راه ← ترکیب direct map و Associative

آدرس دهی لرکت پذیر مجموعه‌ای

Associative _ Cache را به تعدادی مجموعه می‌کنیم

در هر مجموعه به صورت Associative مقادیر را جای دهیم دیگر نیست توی کدام بلاک قرار می‌گیرد

V	T	Block 0	Set 1
V	T	...	
V	T	Block M-1	
V	T	Block 0	....
V	T	...	
V	T	Block M-1	
V	T	Block 0	Set N/M
V	T	...	
V	T	Block M-1	

مجموعه‌ی ۱

مجموعه‌ی N/M

اما دیتای اصلی را توی یک مجموعه‌ای به خصوص که از روش باقی مانده بدست میاد قرار می‌دهیم

Scanned by CamScanner

یعنی ما یک آدرس را که می‌خواهیم از RAM بیاریم توی cache اول می‌بینیم
توی کدوم $\boxed{set}$ جا بدیم (در ایندکس روی باقی مانده)
اما توی این $\boxed{set}$ می‌تونیم توی هر بلوکی که خالیه دیتامون را قرار دهیم

ترکیب دو روش قبلی ← هم احتمال capacity و هم conflict
مشکل

اما کمتر از روش‌های قبلی (خالی یا چندین فقط یا نه) ← کارکرد بهتری داره

مثلاً اگر $\boxed{set\ با\ 4}$ داشته باشیم ← می‌گیم
4 _way set Associative cache
$\boxed{N/M}$ _way " " " "

خلاصه:

	Direct map	set Associative	Fully Associative
Cold	✓	✓	✓
conflict	✓	✓	✗
capacity	✗	✓	✓

چون یک دیتا به یک بلوک ← چرا؟ توی "مستقیم" و Associative خاص یا یک set خاص map می‌شود
اما توی fully نداریم . چرا؟ چون می‌گردیم دنبال یک بلوک خالی به صورت آزادانه باشد ؟!!
توی fully و مشکل capacity را داریم چرا؟ بلوک‌هایش پر شده.
چون بلوک‌ها را آزادانه پر می‌کنیم ← ممکن است پر شود یک جای cache
set
توی Associative هم ممکن است capacity باشد چرا؟ توی هر مجموعه‌ای ممکن است این مشکل پیش بیاد دیتا به صورت یک assign شود و ممکن است سه تا یک

دیرتر بعد ← ولی دیرتر

چیزی گیم لزوماً نیست مجموعه‌ای ارزان‌تر و بازده‌تر است

چون فاصله‌ی بیشتر در set ، کوچک‌تر از فاصله‌ی بیشتر در کل سمت افزایش است ← محدودیت بیشتر کوچک‌تر ← برخلاف حافظه‌ی Associative

که هدف حافظه بزرگ‌تر بشه ،

بیشتر کمتر می‌شه

Session 22

SSD ← برای کدینگ کامل و برنامه‌های اصلی

HDD ، یک حافظه‌ی کند برای ذخیره‌ی داده‌های افزایش

hard disk

توی حافظه‌ی معمولی ← آدرس می‌دیم ، مقدار می‌خوانیم

توی Associative ، نمی‌دونیم مقداری که می‌خواهیم کجا قرار گرفته فقط یک نشونه از مقدار داریم.

مثلاً ، دنبال خانه‌هایی در حافظه می‌گردیم که مثلاً بیت اولش 1101 باشد و بقیش مهم نیست چون‌که بیشتر بقیش هم نیست.

Key register با → 1111

می‌گذاریم

این 4 بیت توی تمام خونه‌های حافظه با مقدار خونه‌ها مقایسه می‌شه و هر خونه‌ای که 4 بیت اولش 1101 باشد ← بیت متناظرش در M ، یک می‌شود

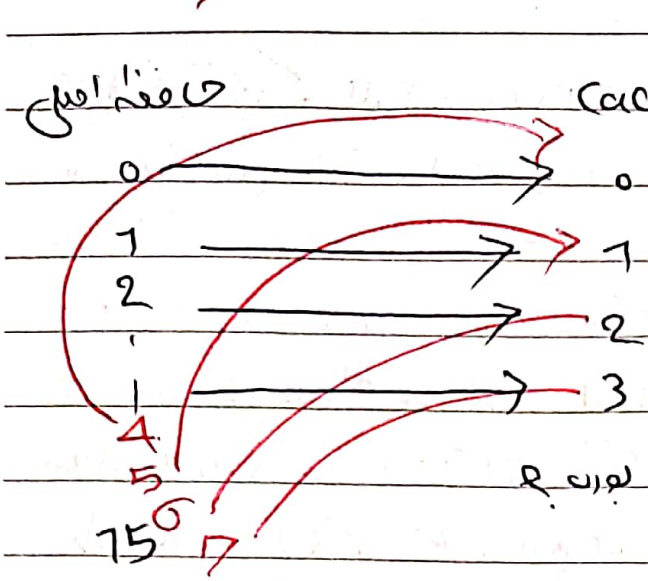
نحوه‌ی ساخت tag ، اگر cache ، آدرس دهی مستقیم داشته باشد

می‌گیم ← m خونه‌ی حافظه‌ی رم را توی n خانه‌ی حافظه‌ی cache جای می‌دهیم

اگر $n=4$ و $m=2$

۱۶ تا بلوک

۴ تا بلوک دارد



0 → 0

4 → 0

8 → 0

از کجا بفهمیم این بلوکی که توی صفحه ۰ کدام بلوک RAM بوده

یک Tag می گذاریم

tag = بقیه ی بیت های آدرس حافظه ← بقیه ی بیت ها کنار را

اگر ۴ بیت آدرس داشتیم ← ۲ بیت شماره ی بلوک را مشخص می کند

offset word

به جز

tag = کل بیت های آدرس ← اگر حافظه تمام آدرس ها بزرگ نیست باشد

چرا کل ۴ حق آزادانه می توانیم جدید کنیم ← مثلا ۰

5 → 0

13 → 0

طبق محلی بودن زمانی

نیاز های جایگزینی اطلاعات

LRU ۰ در گذشته دور ارجاع داشته least recently used

(اخیراً کاری با ش نداشتیم)

FIFO ۰ به ترتیب ورود

LFU ۰ کمترین دفعات ارجاع را داشته باشد

MIN ۰ در آینده کمترین ارجاع را دارد

به دلیل مشکل capacity باید یه بلوکی را بندازیم بیرون

بیرون

LRU ← توی یه آرایه ی ۰ برای هر بلوک توی cache ۰ تایمر می گذاریم

اگر توی بلوک hit رخ بدهد ← تایمر را reset می کنیم

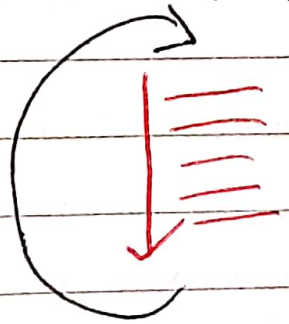
↓ اونی که تایمرش بیشتر بقیه است (خیلی وقته باش کار نکردیم) را می دهیم

آیا مهم نیست که بلوکی که می‌دهیم بیرون باقی کاری ندارم در آینده به آن
کسی نیاز است (احتمال)

این احتمال چطوری می‌تواند به ما کمک کند؟ طبق رفتار برنامه
اگر رفتار برنامه طوری باشد که از بالا به پایین دستورات اجرا شود و دیگر بعد از
مدتی با دستورات بالایی کاری نداشته باشیم ← LRU خوب جواب می‌دهد
↓ لیست‌هایی که به کار می‌بریم باید متناسب برنامه باشد
اگر نه کار را خراب می‌کند تازه

مثال: حافظه ← فرض کن ۵ بلوک حافظه می‌خواهد و
cache ۴ تا بلوک دارد ← برای ۴ تای اول اوکیه اما به نحوی که برای ۵
طبق LRU ← اولی را می‌اندازد بیرون ← حالا دوباره حافظه می‌خواهد
تکرار شد ← با بلوک اول کار دارم ولی miss شد (تا لحظه‌ی پیش بود ۸۸)
↓
الان هیچ‌کدام بلوک دوم را می‌اندازد و به جایش اولی را می‌گذارد و
این داستان ادامه دارد.

بدترین حالت برای LRU



حافظه متوسط زمان دسترسی به یک بلوک = دسترسی به حافظه / تعداد بلوک‌های cache + ε

مقدار فضای که کم می‌آرد

حافظه بهترین کار ← باید به برنامه کمک کنیم
داشته باشیم

لیست باید طوری پیاده‌سازی شود متناسب با برنامه که
max = hit را داشته باشیم. متناسب با رفتار برنامه

خوب نیست

زیاد

۲ حالت ← لیستم
خاص منظوره ← عام منظوره ← لیست برای همه برنامه‌ها لازم‌ها

FIFO ← پیاده‌سازی بارد (خوب یا بد بودن به رفتار برنامه بستگی دارد)

LFU ← least frequently used

کانتر داریم ← هر بار که hit برای بلوک رخ می‌دهد، کانتر بلوک را یکی افزایش می‌کنیم ← موقع جایگزینی

count++

کدام را بندازیم بیرون؟

آدمی که کانتر کمتری دارد

کمتر باشی کار داشتیم

MIN: در آینده کمتر باشی کار داریم (پیشگویی ← predictor داریم)

بر اساس احتمال و طبق رفتارهای قبلی برنامه‌ها

Benchmark های قبلی

بر اساس اطلاعات قبلی که داریم و احتمال، پیش‌بینی می‌کنیم

اطلاعات را به کمک compiler می‌توانیم بدست بیاوریم

برنامه چون از اول تا آخر یکبار قبلاً کامپایل شده

یک دید اطلاعات دارد

به CPU می‌تواند کمک کند که پیش‌بینی خوبی کند

اشکال: پیچیدگی سخت‌افزار ← گران تر تاخیر

سیاست‌های به‌روزرسانی اطلاعات

وقتی CPU یک مقداری را توی Cache تغییر داد ← باید RAM را هم آپدیت کند

① write through: با هر تغییر ← آپدیت Cache و RAM

② write Back: فقط در زمان خروج از Cache آپدیت می‌شود

Scanned by CamScanner

الگوریتم اول ← تاخیر ← هر بار بخواهیم با RAM کار کنیم

مزیت: سازگاری در thread ها و تاخیر بار اول در هسته‌ها

الگوریتم دوم: فقط زمانی که می‌خواهیم evict کنیم ← RAM را آپدیت کن

یعنی طبق locality زمانی و مکانی: کارات را با [n] انجام بده ← نمی‌خواهد RAM را آپدیت کنی

وقتی عبور شدن n را از cache بفرستی بیرون ← RAM را آپدیت کن

ایده خوبه اما همین الان هسته‌ی دیگر که می‌خواهد از [n] استفاده کند آپدیت لازم را نتواند دریافت کند ← در [چند هسته‌ای] ← دستم دچار مشکل می‌شود

پس بر حسب شرایط و نیاز و کاربرد ← تصمیم می‌گیریم کدام سیاست خوب است

حافظه مجازی virtual memory

بعد از RAM: بخشی از حافظه‌ی جانبی را هم به عنوان تغذیه کننده‌ی CPU در نظر می‌گیریم ← چرا؟ ↑ حجم حافظه‌ی تغذیه کننده

چرا RAM را زیاد نمی‌کنیم و می‌بریم از حافظه‌ی جانبی؟ گران است

حجم

ارزان است

این موقعی که میریم سراغ RAM ←

اگر بلوک مورد نظرمون توی RAM بود که خوبه اگر نبود باید بریم از حافظه جانبی (HDD یا SSD) بیاریم توی RAM

* کاربر احساس میکند حافظه زیادی در اختیار دارد
* از حافظه جانبی برای ذخیره‌ی اطلاعات مازاد استفاده می‌کند
* محتوای حافظه مجازی به صورت صفحه به صفحه و قطعه به قطعه به حافظه اصلی انتقال می‌یابد

به جای بلوک بلوک مثل Cache و RAM

حافظه ← ① تغذیه کننده‌ی CPU ✓
② نگهدارنده‌ی اطلاعات (مثل هارد اکسترنال)
و ...

این هم الکی

حافظه تا RAM حساب می‌شه اما یک مشکل داریم و فضای کم RAM
اگر مجموع RAM مورد نیاز برنامه‌هایی که داریم اجرا می‌کنیم > کل RAM سیستم بشه
چه اتفاقی می‌افتد

اگر بیاید توی RAM بیاریم در صورتی که جمع RAM ها کمتر الکی
اضافه می‌کنیم

حل مشکل ← virtual memory ← با یک مدیریتی خوبی، یک حافظه‌ی بیشتر از RAM واقعی را در اختیار برنامه‌های بیشتری می‌گذارد ← کاربر می‌گذاریم
run کند

اگر همیشه بخواهد بره سراغ حافظه جانبی ← hit ratio ↓
virtual memory

مثلاً در windows ← توی درایو C الکی ← وقتی داره پر می‌شه error می‌دهد
در لینوکس ← swap داریم

** پس می‌کنار داده بخشی از حافظه‌ی جانبی کنار RAM تا فضای RAM بزرگتر شه **

* اگر SSD شود خیلی بهتر می‌شود *

Scanned by CamScanner

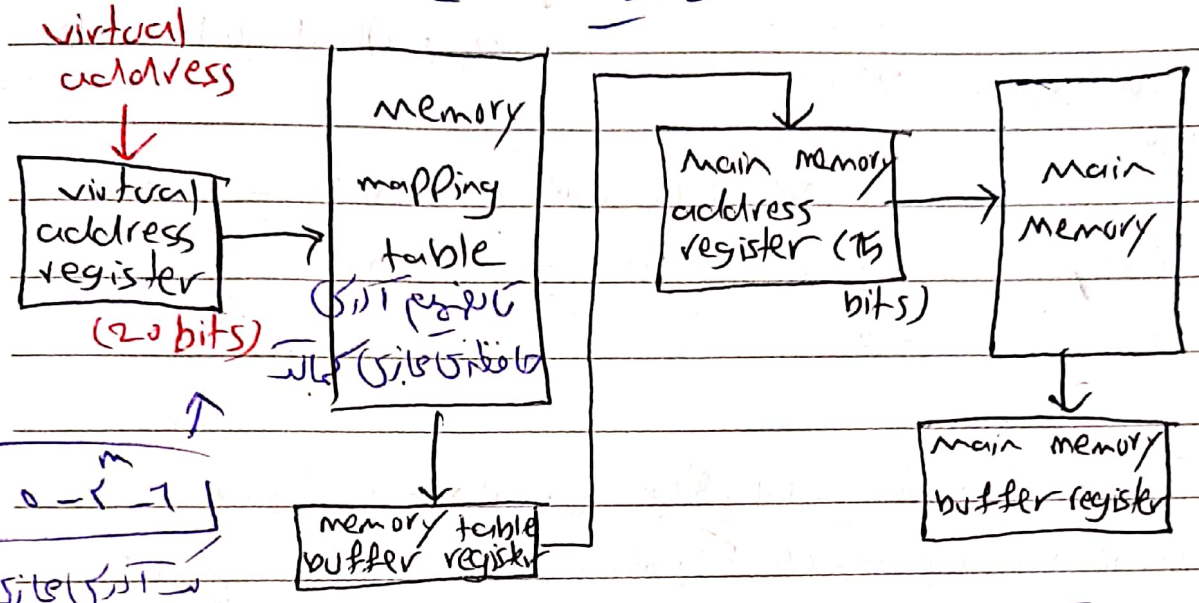
فضای بیشتری در اختیار تا وقتی حافظه جانبی حساب می‌شد

داریم.

RAM 0 — $2^n - 1$

ترجمه‌ی آدرس مجازی به آدرس فیزیکی

CPU وقتی یک آدرس می‌خواهد باید ببینیم کجای این آدرس مجازی قرار گرفته



$0 - 2^m - 1$

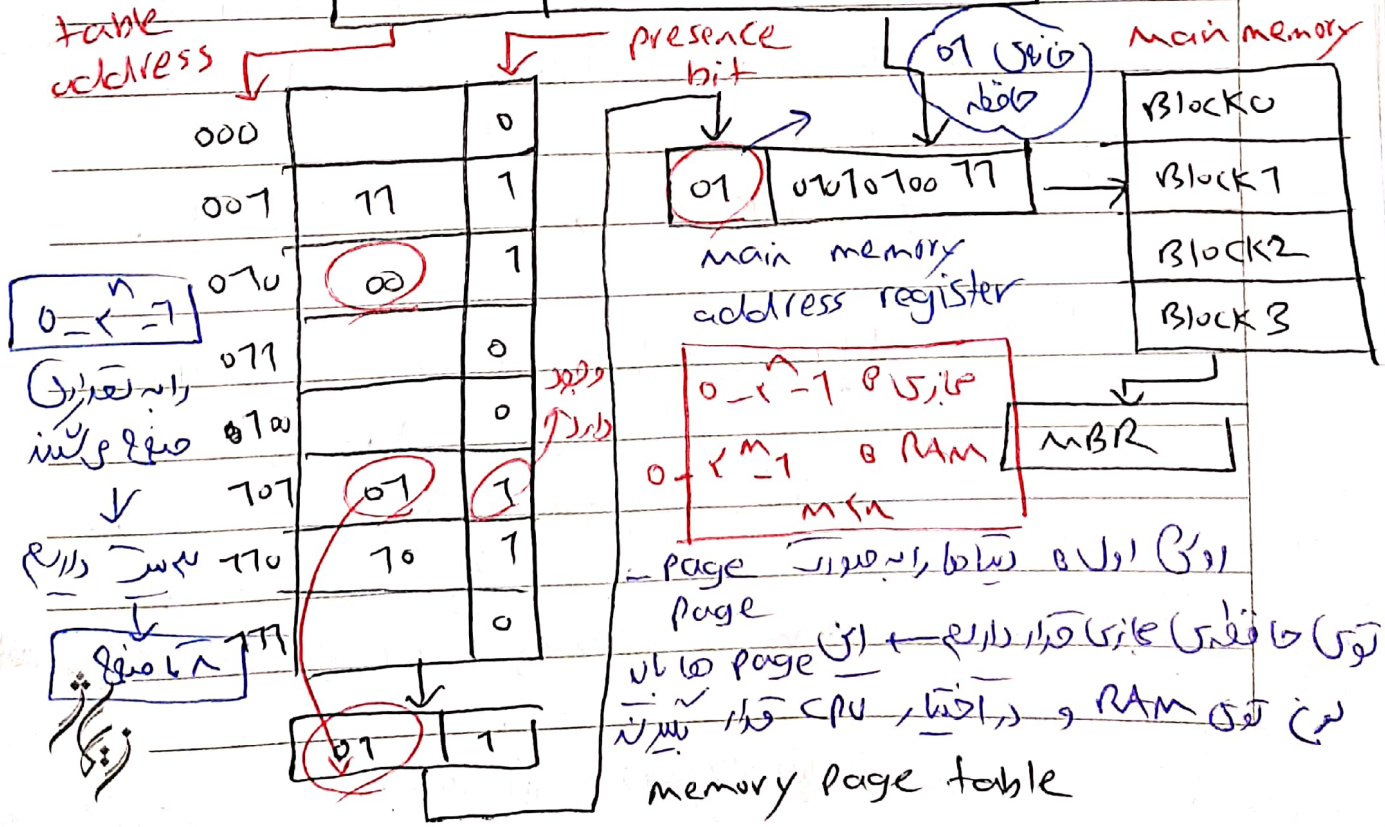
یک آدرس مجازی که CPU اعلام کرده

چون آدرس مجازی داریم ← ترجمه آدرس نیاز داریم

شماره‌ی صفحه ← Page no. — شماره‌ی خط در صفحه ← Line number

1 0 1	010101 00 11

Virtual address

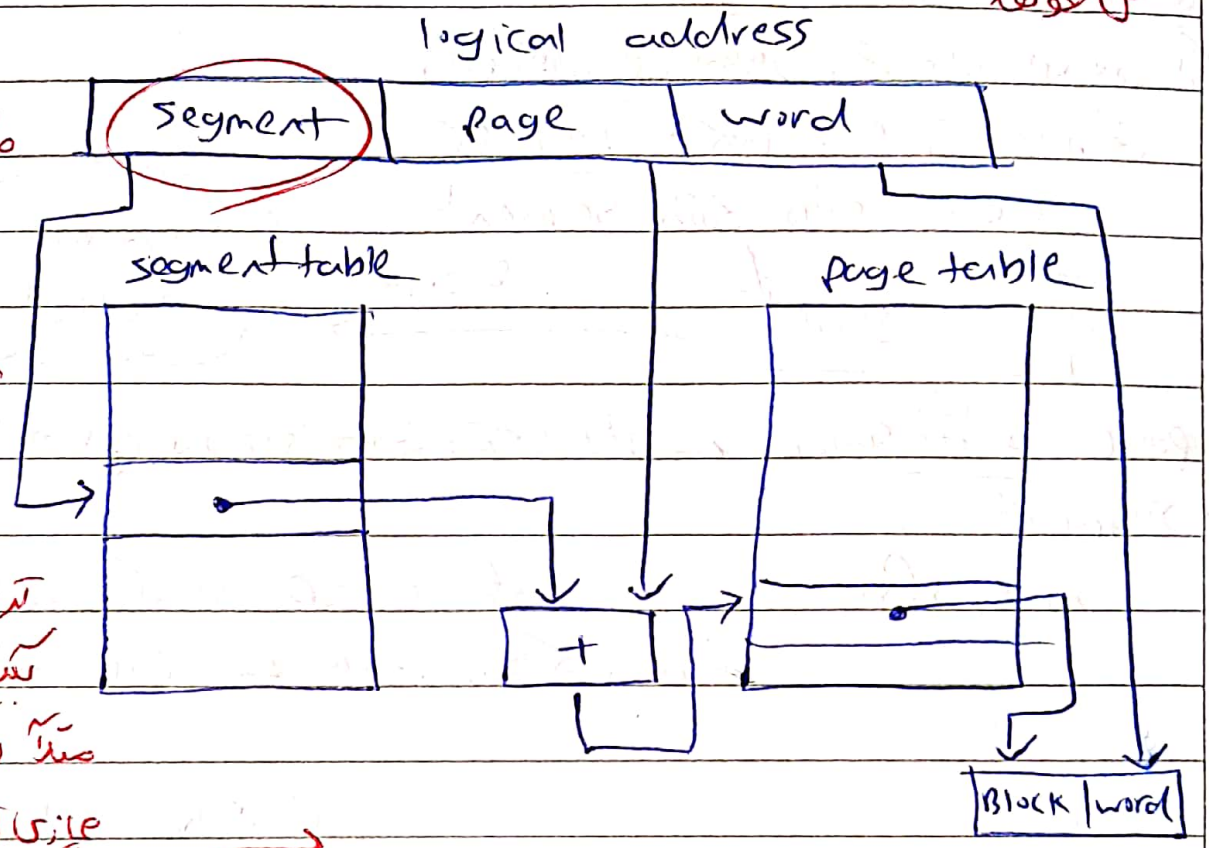


اولین اولین داده‌ها را به صورت page page توی حافظه‌ی مجازی قرار داریم ← این page ها باید توی RAM و در اختیار CPU قرار بگیرند

memory page table

پس برای اینکه یک متغیری مثلا x را پیدا کنیم ← شماره صفحه‌شو باید پیدا کنیم
و ببینیم توی اون صفحه، توی کدام خط، x را داریم

این قابلیت الت اما شماره‌ی page، mapping می‌خواهد



مثلاً حافظه‌ی مجازی ۸ تا صفحه داره ولی RAM، ۴ تا صفحه داره

ترجمه می‌خواهیم که بگیم این صفحه که مثلاً شماره‌ی ۲ حافظه‌ی مجازی الت، در کدام یک از صفحات RAM قرار گرفته.

شماره page را توی جدول آدرس‌ها به صورت lookup دنبالش می‌گرداند
تا بفهمیم صفحه‌ی ۲ در کدام page از صفحه‌ی اصلی قرار گرفته
و آیا اصلاً قرار گرفته الت یا نه (بیت presence)

اگر صفر بود ← می‌رفتیم سراغ حافظه‌ی جانبی

هر صفحه را به تعدادی شکست کوچک می‌شکنیم

علاوه بر صفحه segment هم داریم

حالا هم باید دنبال page بگردیم هم segment

وقتی page page حافظه را تقسیم می‌کنیم به قسمت‌های مساوی
اندازه‌ی page را حتماً باید بگیریم به اندازه‌ی بزرگترین برنامه‌ی ممکن
مثلاً اینجا بزرگترین برنامه 2MB الت و یک برنامه داریم 4KB الت ←
یک فضای ۲ مگی را برای 4K داریم الکی اشغال می‌کنیم ← چرا؟
fragmentation در حافظه ← اتلاف حافظه ← راه حل؟

Scanned by CamScanner

فضای بیشتر در اختیار من می‌گذارد

فایده segmentation

کنار هر segment، یک لیستی اطلاعات در حافظه
قرار می‌گیرد، این اطلاعات می‌گه صاحب این قسمت از کد
کیه (یعنی به کدوم برنامه حق داره به این
segment دسترسی داشته باشه؟)

به اسمی PID

فایده 2

memory protection

هر کسی به هر جای حافظه دسترسی نداره ← هر برنامه فقط به page ها و segment های خودش دسترسی داره

CPU چک می‌کنه اگه PID با info ... مطابق نباشه ← دسترسی رو بلاک می‌کنه.

برای بحث بدافزارها

---

session 23

* تسهیلات ورودی/خروجی یا دستگاه‌های جانبی امکان برقراری ارتباط کامپیوتر با محیط پیرامون را فراهم می‌کنند.
* این محیط می‌تواند در ارتباط با کاربر و یا فرایند صنعتی و ... باشد
* عموماً سرعت این واحدها بسیار کمتر از پردازنده اصلی است.
* برای جبران این اختلاف نیاز به مدار واسط و ارتباط با حافظه است.

کم کردن اختلاف سرعت

مثال‌هایی از دستگاه‌های جانبی

- صفحه نمایش ← صنعتی / گرافیکی
- صفحه کلید
- چاپگر ← ماتریسی / لیزری
- دیسک مغناطیسی

ذخیره‌ی بخشی از حافظه‌ی مجازی روی دیسک مغناطیسی

ندرم افزاری جزو حافظه

سخت افزاری جزو حافظه

بخشی از این دیسک، تکمیل حافظه مجازی

انواع سنسور ها دما و رطوبت و فشار و شدت نور و ...

دستگاه های جانبی
- مبدل آنالوگ به دیجیتال (ADC)
- مبدل دیجیتال به آنالوگ
- الکترود
- پلاتر

مدار واسط
- تبدیل سیگنال ها: دیجیتال به آنالوگ، آنالوگ به دیجیتال
- هم زمان سازی ← سنسرون کردن کامپیوتر و I/O (سرعت ها متفاوت است)
- تبدیل داده ها ← موازی به سریال یا برعکس و ...
- بافر کردن عملکرد

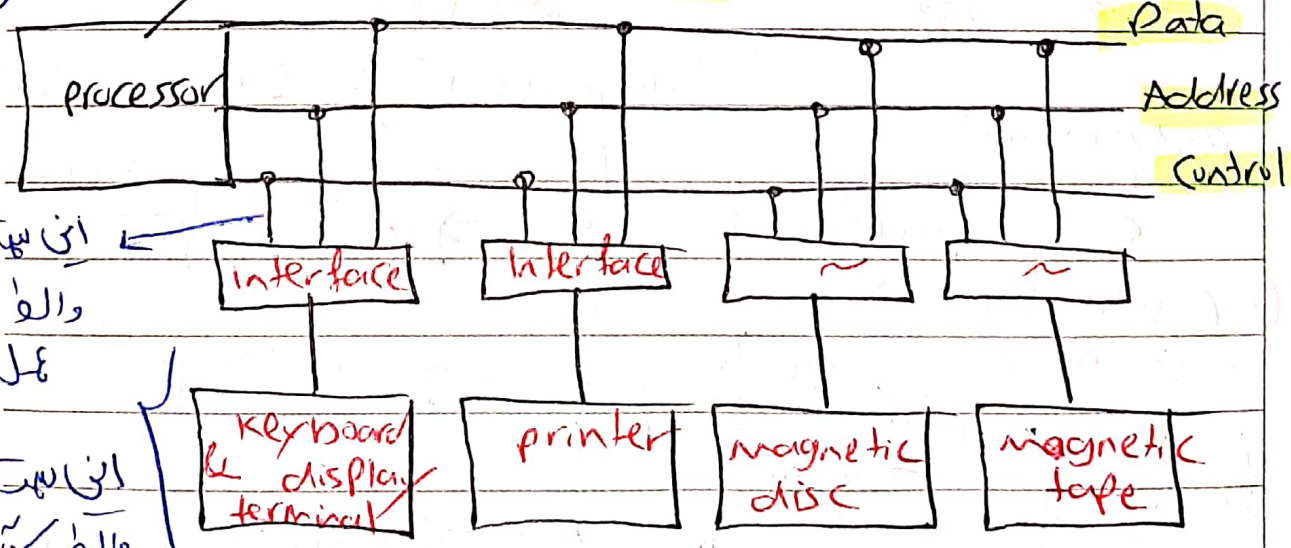
I/O

آنچه دستگاه جانبی می تواند انجام دهد را به کامپیوتر معرفی کند و فرمان های لازم را از کامپیوتر بگیرد

چه زمانی و در چه کدامیک و چه کسی و چه داده ای را بفرستد یا بخواند ← تعیین اصلی

ارتباط از طریق گذرگاه مشترک

I/O bus



این بخش مدارهای واسط مثل هم عمل می کنند

این بخش مدارهای واسط کلا فرق دارد (سرعت و عملکرد متفاوت)

هیچ ارتباطی لزوما شبیه باهم

هر I/O مدار خاص خودش را دارد

واسط

CPU ← هم زمان به IO و حافظه دسترسی دارد.

مزیت ارتباط هم زمان حافظه و IO

شدرها باکی بیشتر تر، حافظه و

راه گذرگاه مشترک پایه های بیشتر تر CPU و سخت افزار بیشتر

ارتباط پردازنده با حافظه و دستگاه جانبی ← گذرگاه داده مشترک (Data) با گذرگاه کنترل جزا و آدرس جزا

گذرگاه داده و کنترل مشترک

مثل کامپیوتر پایه

« ثبات آدرس ورودی / خروجی »

در دو روش اولی ثبات در فضای آدرس جزا ← دستورات جزا برای کار با ورودی / خروجی

خط آدرس IO جدا باشد ← فضای آدرس جدا می شود

در روش دومی ثبات در فضای آدرس حافظه، (خط آدرس IO با حافظه یکی)

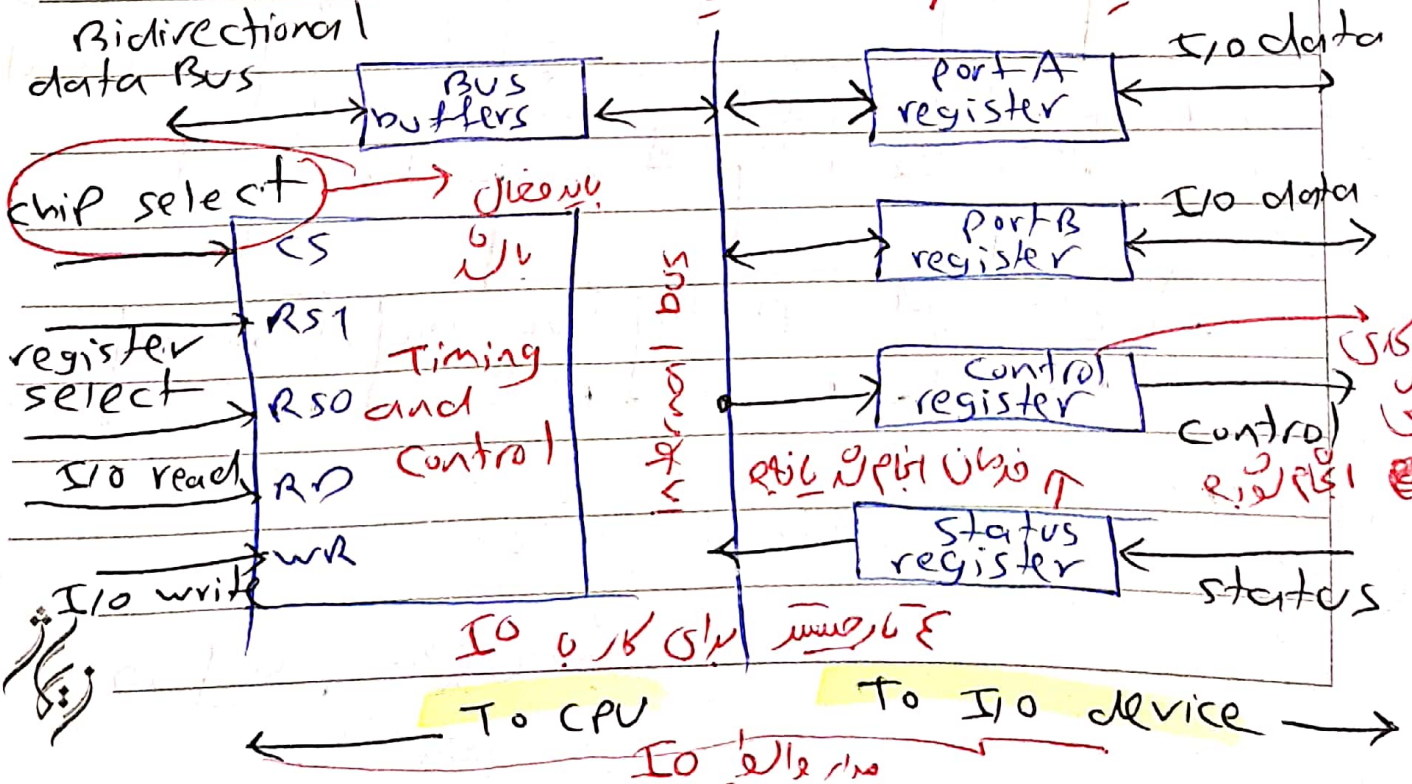
* دستورات یکسان با حافظه

* مدار واسط طبق آدرس انجام شده از سوی پردازنده، دستگاه مورد نظر را فعال می کند

memory mapped IO

فقط آدرس فرق می کند

آدرس صفر IO ≠ آدرس صفر حافظه



کد آدرس این نیز

address	CS	RS1	RS2	عملکرد
0xCFA0	0	x	x	هیچ ثباتی انتخاب نمی‌شود
0xCFA0	1	0	0	انتخاب ثبات پورت A
0xCFA1	1	0	1	B ~ ~
0xCFA2	1	1	0	کنترل ~ ~
0xCFA3	1	1	1	وضعیت ~ ~

عملکرد مختلف

** CPU وقتی با یک دستگاه I/O کار دارد باید پایه‌ی Chip select را فعال کند

اگر نباشد ← CPU اصلاً با دستگاه مورد نظر کار نمی‌کند

مثلاً اگر $\boxed{CS}$ فعال شد ← RS1 و RS0 را بسته به اینکه با کدام register کار دارد فعال می‌کند

address : $\boxed{0xCFA0}$ → آدرس دستگاه مورد نظر →

این دستگاه ۴ تا آدرس می‌خواهد برای ۴ تا رجیستری که داریم

CS : $\boxed{1100111110100000 \mid xx}$

۱۴ بیت — ۲ بیت

IO حتی آدرس‌ها می‌تواند باشد

۱۴ بیت ← می‌آید کنترل CS — And

And توی مدار والط

CS=0 ⟷ ~ کار نداریم ← صفر شد

CS=1 ⟷ با دستگاه کار داریم ← یک شد

آدرس با فلیپ فلاپ برنامه‌ریزی می‌شود

۲ بیت آخر: 00, 01, 10, 11

Scanned by CamScanner

یک بیت می‌گذاریم و داده‌ی ۱۶ بیتی (۲ بایت) یک بیت یک بیت می‌فرستیم ← سریال

A —— B

session 24

انتقال داده

نشست دهم

توازی در ارسال ← موازی ← پارالل

انواع تبادل اطلاعات

A —— B ۱۶ bus

همزمان ← هم‌زمان

لنگرون بودن ← غیرهم‌زمان

سریال یا موازی کدام بهتر؟

✓ سریال ← نویز کمتر می‌گیره بهتر است

✗ موازی ← سرعت بیشتر ← تعداد لیدهای بیشتر ← قیمت بیشتر

در ظاهر سرعت بیشتر

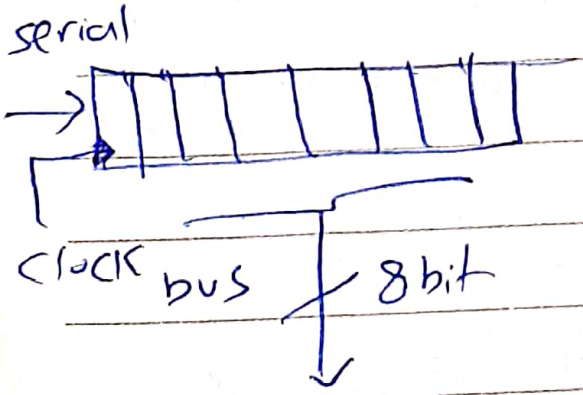
۱۶ تا لید داریم که داده ۱۶ بیت را موازی می‌فرستن ← حرکت‌ها از صفر به یک تغییر می‌کنند ← حرکت القای مغناطیسی ایجاد می‌کنند ← داده‌ی هم را خراب می‌کنند و نویز محیط هم هست

هم خودشان هم را خراب می‌کنند هم محیط

موازی دیگه استفاده نمی‌شود ← داده‌ی لازم دارد مقصد می‌رسد

سوال: از یک داده‌ی دودویی به صورت سریال گرفتیم و می‌خواهیم توی CPU تبدیل کنیم به داده‌ی ۸ بیتی پارالل

با کمک shift register ۸ بیتی



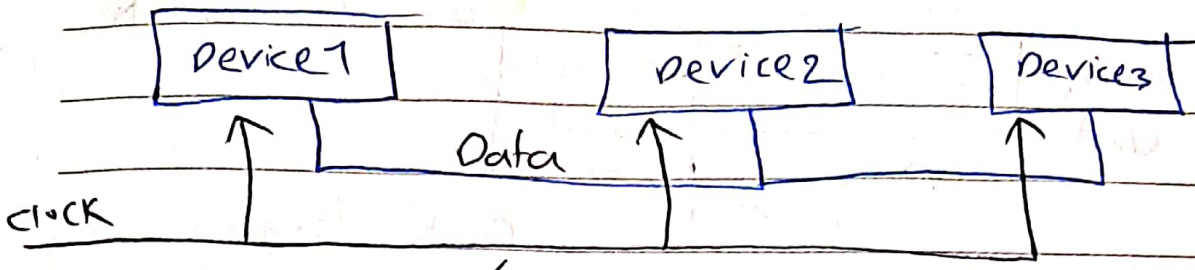
۸ تا کلاک صبر می‌کنیم تا داده کامل بیاید توی رجیستر

در کلاک ۹ ← داده را به بایت ۸ بیتی تحویل می‌دهیم

Scanned by CamScanner

«تبادل داده همزمان»

هر ارسال داده با یک پالس ساعت انجام می‌شود.

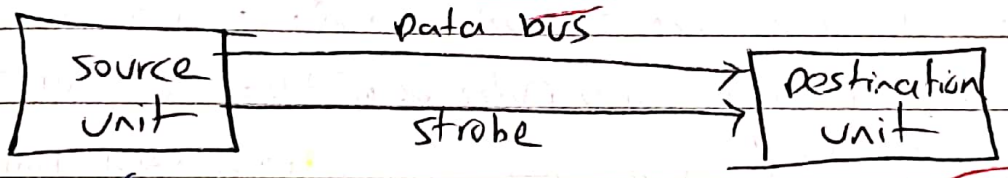


← ضمن امکان تاخیر و ایجاد نویز در سیگنال پالس ساعت

«ارسال داده غیرهمزمان»

* هر دستگاه پالس ساعت خودش را دارد.
* سرعت کار دو دستگاه می‌تواند متفاوت باشد

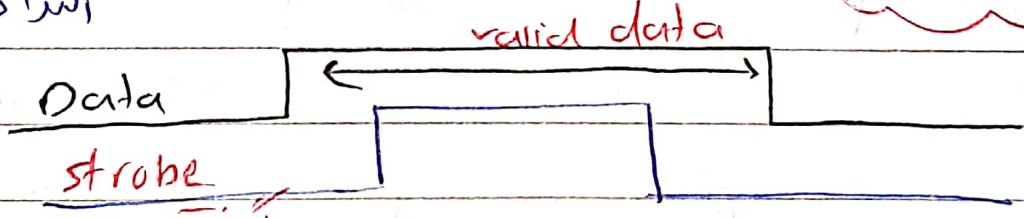
روش کنترل ارسال با سیگنال strobe



مبدأ و مقصد با هم اشتراک می‌گذارند

Block diagram

کنترل از سمت مبدأ



حقیقی دیتا را روی خط گذاشته داریم ← اگر مقصد دیتا را برداشت چی؟

Timing diagram

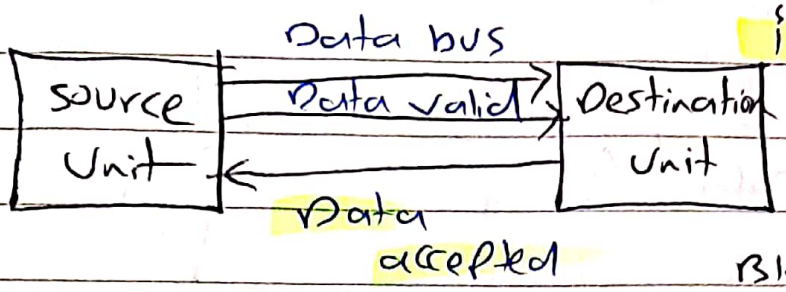
مبدأ دیتا را قرار می‌دهد. بعد از اینکه دیتا روی خط قرار گیرد، مدتی بعد، سیگنال strobe را فعال می‌کند ← این سیگنال به مقصد نشان می‌دهد که از روی خط می‌تواند دیتا را بردارد ← دیتا را برمی‌دارد و مبدأ هم به اندازه‌ی مشخصی strobe را یک نگه داشته ← بعد از اون زمان strobe را صفر می‌کند و می‌تواند یک دیتای دیگه روی خط بگذارد

مشکل ← هیچ فیدبکی از مقصد دریافت نمی‌کنیم ← کنترل کاملاً از سمت مبدأ است

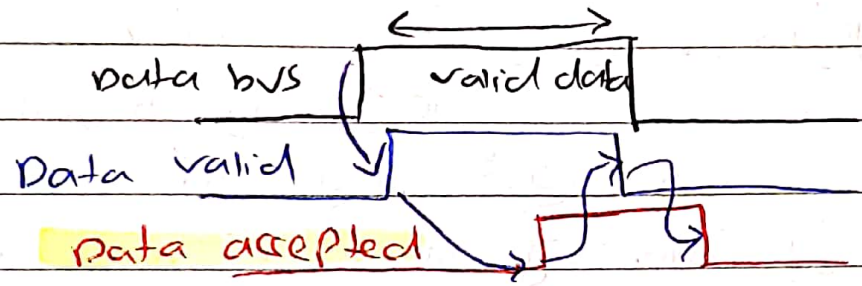
Scanned by CamScanner

روش کنترل ارسال با سیگنال handshake

* کنترل از سمت مبدأ



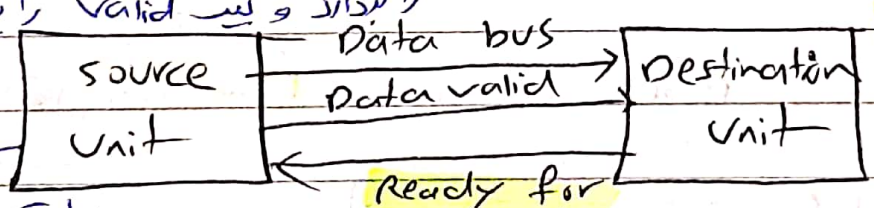
Block diagram

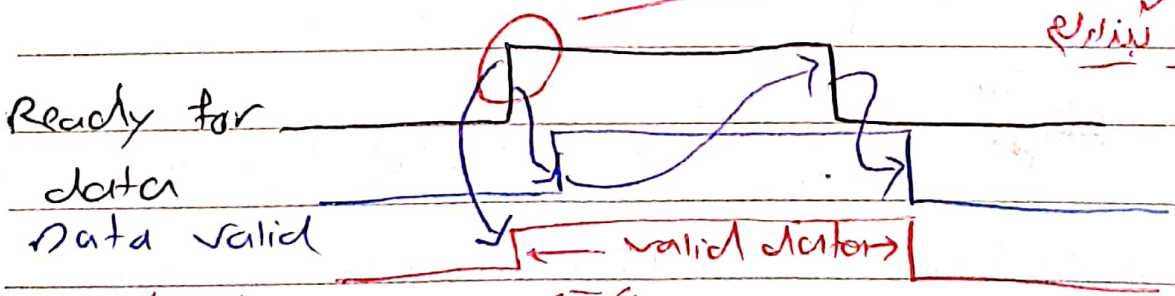


Timing diagram

روش کار: مبدا دیتا را روی باس می‌گذارد و بلافاصله valid را 1 می‌کند تا مقصد بفهمد که می‌تواند دیتا را بردارد. مقصد هر موقع دیتا را برداشت و مطمئن شد دریافت کرده، سیگنال Data accepted را 1 می‌کند ← حالا مبدا می‌تواند دیتای جدید را بردارد و بیت valid را مجدد پایین آمدن تسهیم برای تبادل بعدی

کنترل از سمت مقصد





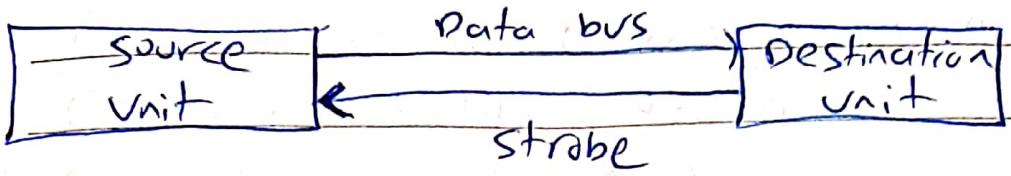
یک بار 0 می‌توانیم دیتا جدید بگذاریم

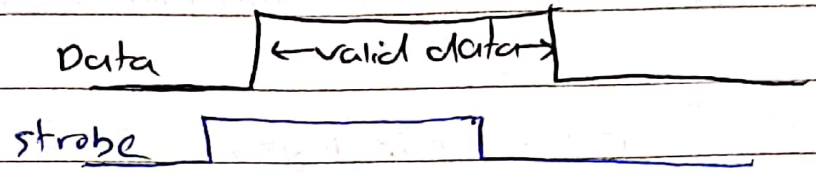
که مقصد دیتا را دریافت کرد

روش کار: مقصد اعلام آمادگی می‌کند که من می‌توانم دیتا را دریافت کنم بعد مبدا دیتا را روی خط می‌گذارد و می‌گوید می‌توانی دیتا را برداری. مقصد دیتا را دریافت کرد، دیتا valid را میاره پایین ← این طوری می‌فهمد که از یکی Parity دریافت بوده و مبدا مطمئن می‌شود

روش کنترل ارسال با کنترل strobe

* کنترل از سمت مقصد: (برای اعلام کنترل از سمت مبدأ)





نحوه:

زمانی که مقصد کاری ندارد ← strobe را فعال می‌کند ← مبدأ می‌فهمد که مقصد آزاد است ← با یک پالس strobe، مبدأ داده‌ای را قرار می‌دهد تا کی این داده روی خط می‌ماند؟ تا زمانی که مقصد strobe را بیاورد پایین، یعنی که داده دریافت شد ← در آن زمان مبدأ می‌تواند داده‌ای را بردارد.

ولی همچنان مطمئن نیستیم که داده توسط مقصد درست دریافت شده. خیر!

چون مسیر نویز دارد راه حل تشخیص نویز در مقصد به بیت توازن Parity

↓

راه حل: اضافه کردن کنترل handshake

اگر فرکانس پالس ساعت دو طرف فرق کند، مثلاً CPU با سرعت 3G و کارت با 100M (صف) ← نیاز به صف داریم برای هماهنگی سرعت‌ها (جبران اختلاف سرعت)

الایر 19 ← شکل سخت‌افزاری

بافر

حال بافر برای اینکه مبدأ چند تا داده هم زمان می‌تواند بفرستد و صف، کمک کند تا خوانش متفاوت است، اولین ورودی به صف مقصد حتی حال اولین خروجی می‌کشد → تایید دریافت کند

بین مبدا و مقصد

صف در مدار واقعا IO پیاده سازی می‌شود / کدام دو طرف فرق می‌کند

تبادل اطلاعات کامپیوتر با ادوات ورودی / خروجی

1. روش برنامه ریزی شده (کدکش) Polling
2. وقفه در یک وقفه کلی در کامپیوتر پایه بود
3. ارتباط مستقیم با حافظه (DMA)

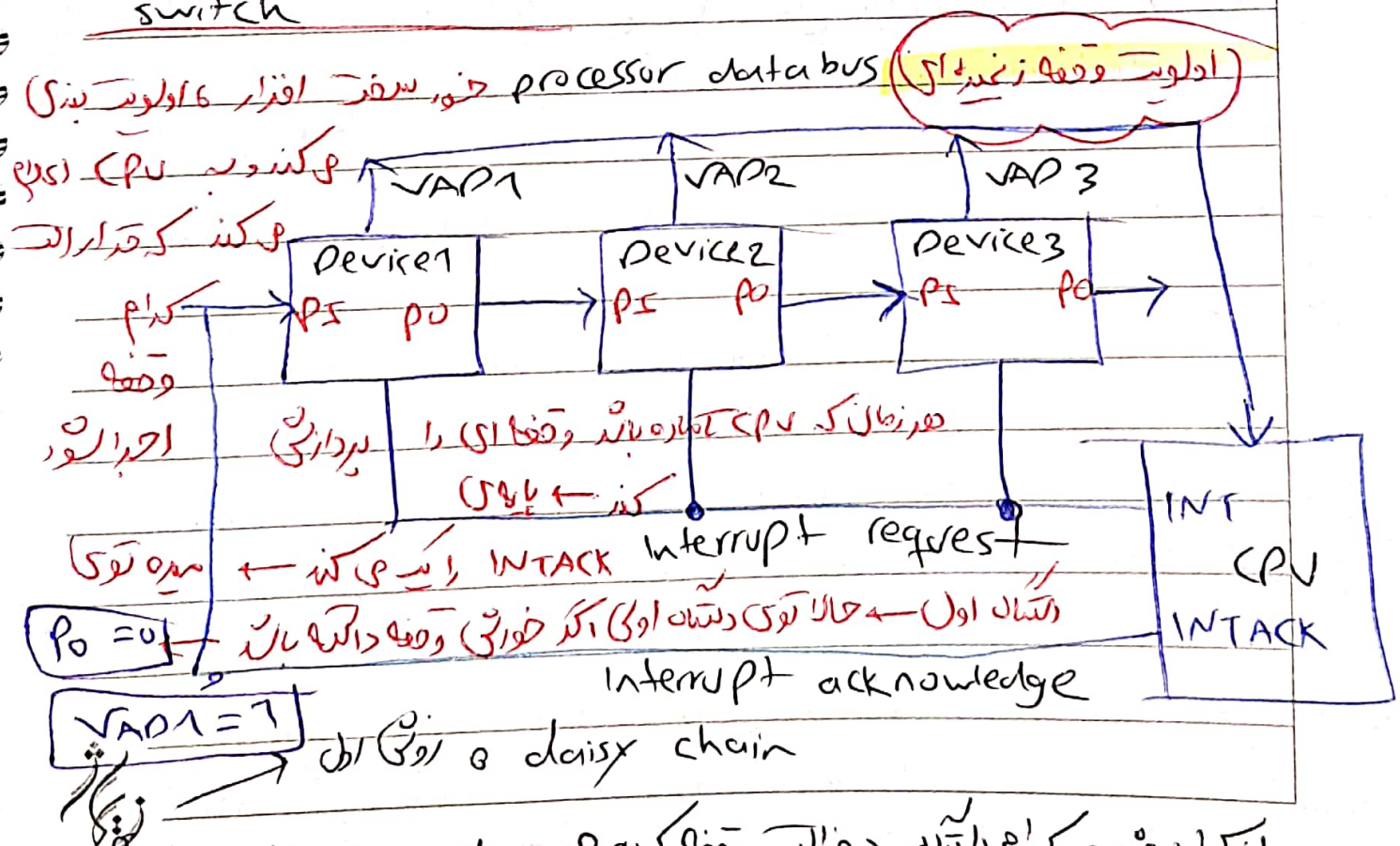
باید بتوانیم چندتا وقفه را پردازش کنیم و اولویت وقفه هم مهم است

اولویت وقفه ← تعیین می‌کند بین دو یا چند دستگاه که همزمان تقاضای وقفه کرده‌اند، کدام یک ابتدا سرویس بگیرد.

۲ نوع ← سخت افزاری ← case ها با اولویت بالاتر، اول نوشته می‌شوند

نرم افزاری ← ترتیب if و else ها یا case های switch

اولویت وقفه زنجیره‌ای



از کجا بفهمیم کدام دستگاه درخواست وقفه کرده؟ ← باید در سخت افزار انجام شود

PI = 0

چون PO = 0 شد ← Device دوم حق درخواست دادن وقفه را ندارد ←

VAD2 = 0 و ـــــ

دستگاه فعال بعدی هم همین طور

دستگاه سمت چپی = بالاترین اولویت و دستگاه سمت راست = پایین‌ترین اولویت

حالا چون ۳ تا دستگاه داشتیم ← $\boxed{600}$ وارد CPU می‌شود ←

Decode می‌کند ← می‌فهمد که بردار وقفه Device اول، وقفه داشته

توی switch case ← وقفه‌ای مربوط به دستگاه اول را اجرا می‌کند

حالت بعدی اینکه $\boxed{1}$ وارد دستگاه اولی شود اما وقفه نداشته باشد ← 1 را به خروجی می‌دهد و $\boxed{VAD_1 = 0}$ ← اگر دستگاه دومی وقفه داشت ←

$\boxed{\text{بردار وقفه} = 650}$ و ...

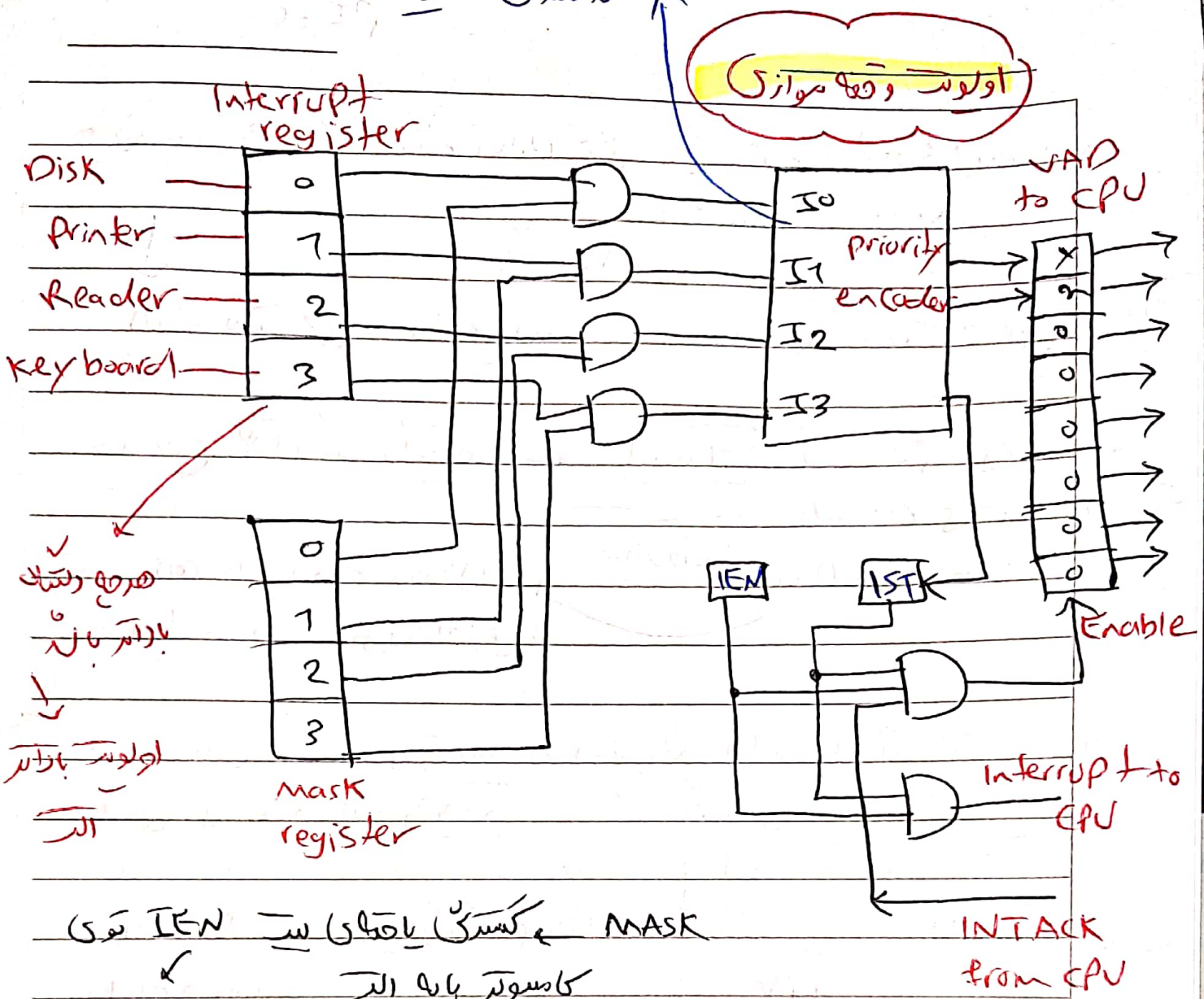
بردار وقفه ← مشخص می‌کند کدام دستگاه وقفه داشته و متناسب با آن CPU کدوسی دهی انجام می‌دهد.

ساختار هر دستگاه ← $\boxed{\text{در اسلاید ۲۲}}$

اولویت وقفه موازی

به هر منبع وقفه‌ای یک بیت ورودی اختصاص داده می‌شود

کد کننده‌ی اولویت دار



MASK ← گسترش یافته‌ی بیت IEN توی کامپیوتر پایه الگ

interrupt enable ⟶ به تعداد دستگاه‌ها 8 بیت IEN داریم

دو بیتی دهی به وقفه را فعال و غیرفعال می‌کرد

هر دستگاه را مستقلاً می‌توانیم فعال یا غیرفعال کنیم.

VAD : vector address

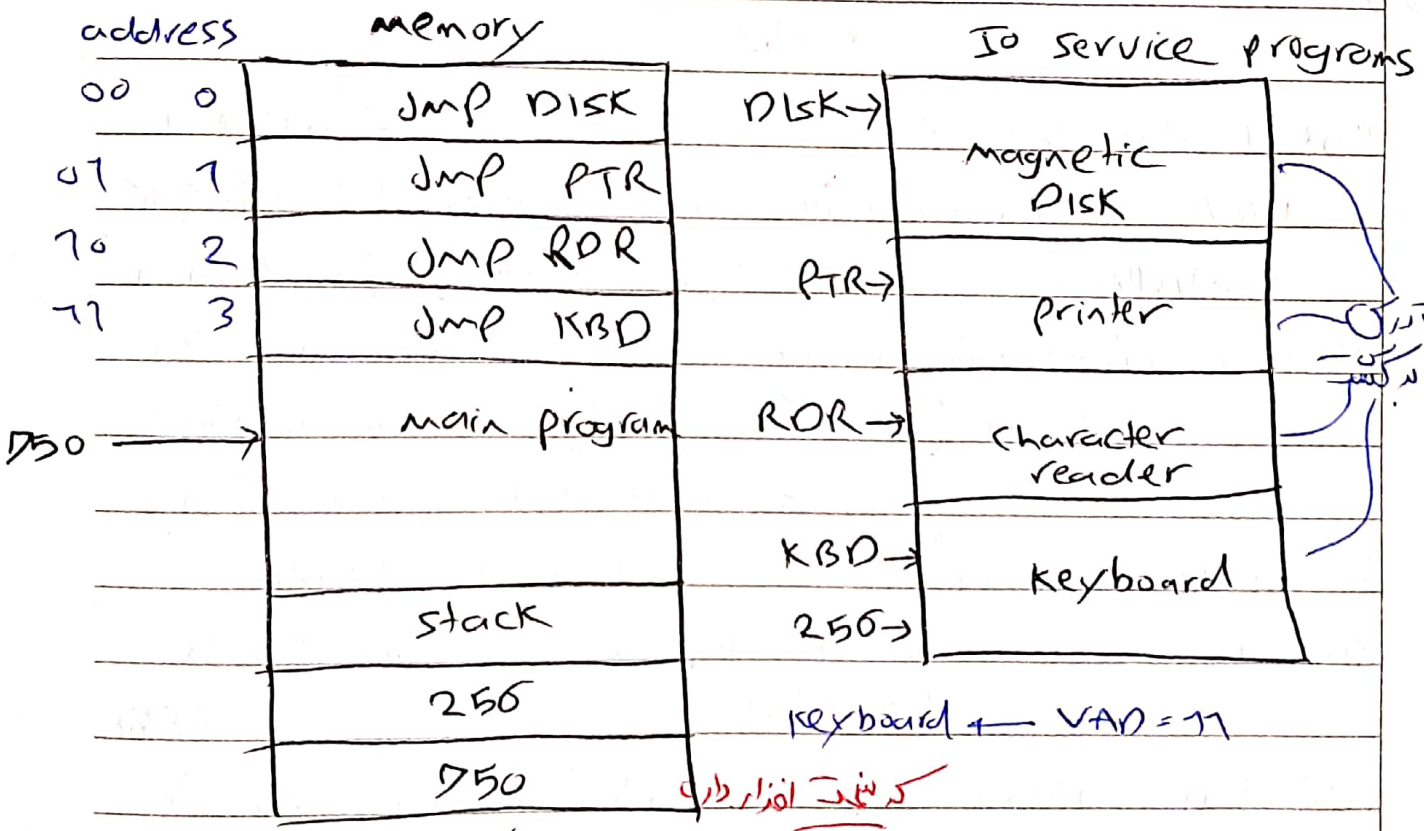
اولویت کدینگ به یک شماره که شماره همان VAD الگ و بدلایلی این CPU می‌خواهد کدام وقفه را باید اجرا کند در غیر این صورت مشغول می‌شود تا CPU آزاد شود و بتواند پردازش وقفه را شروع کند

IST که یک شود به لحظه که وقفه بالادستی هم فعال باشد CPU ← چه کار می‌کند وقفه را تشخیص می‌دهد و در این زمان اگر آماده پردازش دستگاه باشد ← گروه به

مشکل این مدارهای اولویت ؟ در هر دو ممکن است گرسنگی رخ دهد.
یعنی تازه‌هایی که دستگاهی با اولویت بیشتر وقفه دارد، نوبت به هیچ دستگاه دیگری
نمی‌رسد

روش نرم افزاری وقفه

1. وقفه‌ها با اولویت های پایین تر نیز فعال می‌شوند
2. محتوای ثبات ها ذخیره می‌شوند
3. آدرس وقفه ابلاغ می‌شود ← متناسب با vector address
4. محتوای ثبات ها بازگردانده می‌شوند
5. امکان وقفه با اولویت پایین فعال می‌شود



کد نرم افزار دارد

توی آدرس های مشخص حافظه بر اساس VAD، یک switch case می‌نویسیم / خانه‌های اول حافظه را متناسب با VAD قرار می‌دهند
اگر VAD = 00 شد ← به خانه‌ی صفر حافظه می‌رویم ← پرش به
آدرس اولین وقفه‌ی دیسک
VAD = 01 ← خانه‌ی 1 حافظه ← وقفه‌ی پرینتر و

انتقال مستقیم اطلاعات ورودی/خروجی به حافظه

CPU، کنترل کننده‌ی اصلی سیستم است.

ورودی، لزوماً یک دیتا نیست می‌تواند یک باخبری از دیتاها باشد

روی باس دیتا، علاوه بر اینکه CPU فرایندها را کنترل می‌کند، یک واحدی به اسم DMA (direct memory access) اضافه می‌کنیم

کار این DMA به وقتی CPU می‌خواهد یک دیتای زیادی را از یک متغیر روی حافظه بفرستد روی یک device یا از device بفرستد روی حافظه، فرایند را به DMA controller می‌سپارد و DMA، انتقال را

به هر تعداد که لازم است انجام می‌دهد و CPU هم توی این فاصله داره کار خودش را انجام می‌دهد و فقط وقتی فرایند تمام شد، DMA controller یک وقفه به CPU می‌دهد می‌گه کاری که درخواستش را داده بودی تمام شد ← CPU می‌تواند وقتش را صرف کارهای مهم‌تری کند و مکث در گیر کار با دیسک که [کند] است نشود.

به این کار می‌گوییم DMA (انتقال مستقیم حافظه)

(انتقال جانبی به جای اینکه دیتاها را به CPU بدهد، CPU توی حافظه ذخیره کند و برگردد، مستقیم با DMA controller به حافظه می‌تواند دیتا را پیدا کند.

arbiter ← در هر لحظه کدام یک از CPU و DMA، باس را کنترل می‌کنند

تعیین اینکه

SESSION 25

عملیات در کامپیوتر

مهم‌ترین وظیفه کامپیوتر، انجام عملیات و پردازش داده‌هاست.

اگر چه لازم است لزوماً مهم‌ترین بخش یک کامپیوتر واحد پردازش و عملیات نباشد.

بایت و بیت، Xor، or، And، ...

عملیات

زمان کنونی، جمع و تفریق، ضرب و تقسیم، عملیات اعشاری، توابع

کلاک متناسب با کندترین بخش سیستم

عملیات در صنعت

باید طوری باشد که بتوانند عملیات مشخص را انجام بدهند

تا خیلی بخش‌ها کنار یکی نیست ولی چون کلاک مشترک داریم

طراحی سیستم

در سیستم همزمان، دوره تناوب پالس ساعت متناسب با کندترین عملیات تعیین می‌شود

همیشه باید بخش‌ها معطل کندترین واحد خواهند بود

سیستم غیرهمزمان پیچیده و پرهزینه خواهد بود

در طراحی پردازنده بالدرنگ و فرکانس بالا، سادگی و کارایی بیشتر مدنظر است.

عیب، ممکن است ۶۰ تا دستور جمع داشته باشیم و یه ضرب، اما کلاک سیستم بر اساس ضرب تعیین می‌شود ← ۶۰ تا کلاک اضافه معطل می‌شویم چون کلاک ثابت است

برای اینکه بخش‌های سریع‌تر معطل نشوند، به اونها کلاک سریع‌تر می‌دهیم و به اونهایی که کندتر هستند، کلاک کندتر می‌دهیم ← فرکانس کلاک‌ها متفاوت است

عیب، طوری هم زمان کنیم اگر مثلاً ۵ تا ۵MHz خواهند با هم کار کنند، داده‌ها رو که ... می‌گذاریم ← عیب، جمع و ضرب‌های زیاد با فیبها

با فید

زیاد

↓

«↑ پیچیدگی سیستم»

نسخه ← لازم‌تر ولی ممکن است کندتر باشد

اکسکرو ← احتمالاً سریع‌تر جواب می‌دهد، اتلاف وقت کمتر دارد اما پیچیدگی بالاتر می‌رود و هزینه زیاد می‌شود

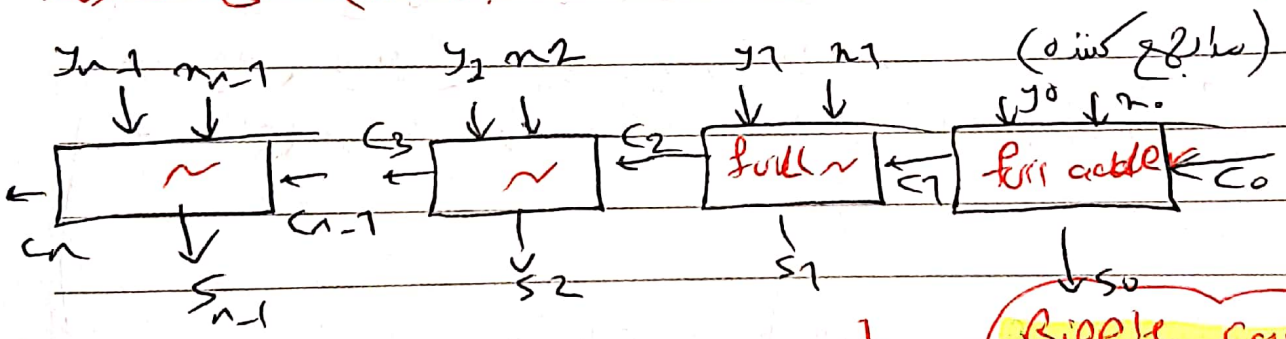
جوری هم سخت‌افزار ساده باشد هم اتلاف کم داشته باشد

راه حل ساده: کم کردن اختلاف سرعت‌ها. (یعنی صدا جمع‌کننده و ضرب‌کننده)

راه کارها:
1. نزدیک کردن تاخیر همه‌ی بخش‌ها و واحدها تا حد ممکن
2. ↑ فرکانس کاری سیستم
3. ↑ راندمان سیستم

برای بالا بردن سرعت پردازش

(اونی که کلاک می‌خورد، راندمان بیشتری به ما می‌دهد) اگر کلاک نزنه.



Ripple carry Adder

وقتی x و y را توی ورودی‌ها گذاشتیم، full adder ها باهم شروع به کار می‌کنند

ولی وقتی هشت داره کار می‌کنه، c_1 که هنوز حساب نشده

ممکن است تغییر کند ← زمانی نیاز است که به اندازه‌ی تاخیر صدا full adder اولی صبر کرده باشیم تا c_1 حساب بشه و فول ادر دوم بتواند نتیجه را اصلاح کند ← زمان می‌برد تا ثابت بشه اول کار برسد به آخر کار

اگر صدا تاخیر هر کدام از باکس‌های بالا $\boxed{+}$ باشد، اگر ورودی‌ها را یکهو عوض کنیم و در لحظه‌ی + نگاه کنیم به جواب‌ها، خروجی مقدار دارد اما

carry ها

این مقدار مثل این است که carry ها وصل نبودند (صفر هستند همشون)

باید به اندازه‌ی $\boxed{n+1}$ صبر کنیم تا carry ها حساب شوند و بیت های S مقدار نهایشون را ببینند

مثال بدترین حالت: تمام بیت ها یک دارند

تاخیر: carry از اول تا آخر

$$\begin{array}{r} 1111 \\ 1011 \\ + \quad 0101 \\ \hline 10000 \end{array}$$

carry 4 صرفه

$\boxed{4t}$ تاخیر

باید به اندازه‌ی تمام carry ها صبر کنیم تا مطمئن شویم تمام بیت های دارند ← بدترین حالت

ex:

$$\begin{array}{r} 111 \\ 1101 \\ + \quad 0101 \\ \hline 10010 \end{array}$$

تاخیر این مدار = $\boxed{2t}$ چون بیشترین carry که تولید شده 2 مرحله بود

برای طراحی سیستم باید بدترین حالت را در نظر بگیریم چرا؟ تا بتوانند در همه‌ی شرایط جواب درست را بدهد. این بدترین حالت توی ripple carry adder با بزرگ شدن عرض جمع کننده بزرگ می‌شود.

4 بیتی ← بدترین حالت $4t$

8 ← 8 ← $8t$

16 ← 16 ← $16t$

چطوری جلوی رشد این تاخیر را بگیریم و یک جمع کننده‌ی کارامد داشته باشیم؟

طراحی مداری به اسم جمع کننده با پیش بینی بیت نقلی

p_i ← انتقال به بیت بعدی

$\boxed{x_i + y_i = 1 \text{ اگر و تنها اگر } p_i = 1}$

دو سیگنال تعریف می‌کنیم

g_i generate

$\boxed{x_i + y_i = 2 \text{ اگر و تنها اگر } g_i = 1}$

در بیت iام، carry

مستقل از شرایط قبلی تولید می‌شود به شرط اینکه $\boxed{x_i \text{ و } y_i}$ هر دو 1 باشند

یا از بیت ها یک باشد

بیت نقلی مرحله‌ی قبل generate کند (یا)

محاسبه‌ی بیت نقلی در هر مرحله‌ی

$$C_i = g_{i-1} + C_{i-1} P_{i-1}$$

مرحله‌ی i ام carry خواهد داشت اگر

یا

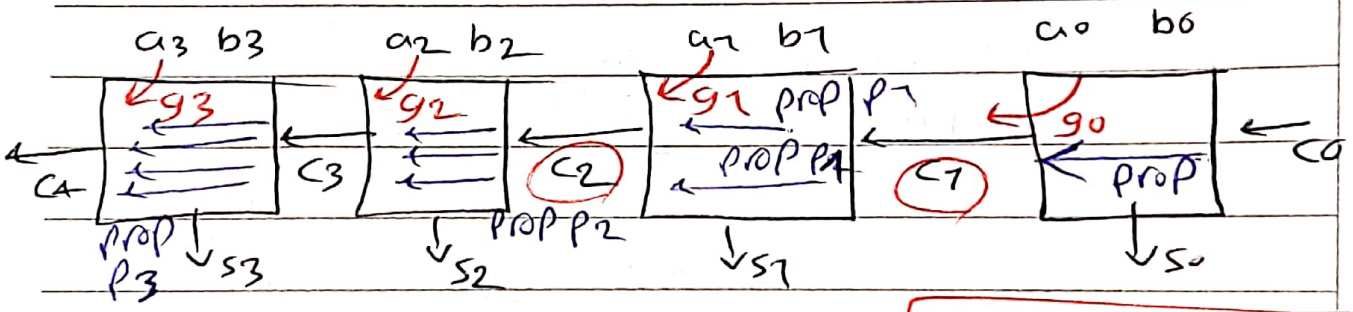
propagate کننده carry که بیتی وارد شود

$$= g_{i-1} + \underbrace{(g_{i-2} + C_{i-2} P_{i-2})}_{C_{i-1}} P_{i-1} =$$

$$g_{i-1} + (g_{i-2} + (g_{i-3} + C_{i-3} P_{i-3}) P_{i-2}) P_{i-1}$$

$$= \ldots\ldots$$

مثلاً C_8 را → تا وقتی که برسیم به C_0 می‌توانیم پیدا کنیم با یک مسیر دیگر



$$C_1 = g_0 + P_0 C_0$$

$$C_2 = g_1 + P_1 g_0 + P_1 P_0 C_0$$

$$C_4 = g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0 + P_3 P_2 P_1 P_0 C_0$$

$$C_3 = g_2 + P_2 g_1 + P_2 P_1 g_0 + P_2 P_1 P_0 C_0$$

مدار جمع کننده‌ی ۴ بیتی ← ۱۶ دروازه ← بیشترین تاخیر ما ۴ تا گیت

Ripple carry

AND تا ۲

XOR تا ۲

۸ تا گیت را ۴ تا گیت کردیم

این مدار پیچیده‌تر است ← اگر مدار یکی از ۴ بیت نباشد ← اگر به دنبال خودش
چون خاصیت بزرگ بودنش در ازای پیچیده‌تر و گران بودنش از دلت
می‌رود

↓

۴ بیت ۴ بیت این مدار را می‌سازیم ← تعادل بین این دو تا و روش عملی

مدار ضرب کننده ← اگر سمت افزار ضرب نداشتیم ← یک قطعه باید بنویسیم که هی
کیفیت بدهد و جمع کند ← همین ایده‌ی کیفیت بدهد و جمع بزند را
سمت افزاری کردند

الگوریتم ۱۷

در حالت داری multiplier و هر تعدادی بیت داشته باشد
به همین اندازه جمع ← به همین اندازه کاغذ داریم
داریم

به ۶ تا داره توی ← multiplier = 6066

به کاغذ کاغذ جمع

ایده ← ↓ تعداد جمع زدن‌ها ← ↑ کردیم

با کدکردن ضرب فیلم به صورت خاص، تعداد عملیات جمع را کاهش می‌دهیم

ex: $\begin{array}{r} 10010777 \\ \times\ 10777770 \end{array}$ ← (عدد دوم ۶ تا) ← ۶ تا جمع
۶ داره
داریم

$A \times (2^7 + 2^5 + 2^4 + 2^3 + 2^1 + 2)$

Shift تا ۶ و جمع داریم

بیت ۱ به بیت چپ اضافه کردن ← هی جمع زدن

2^{n+1}

زیگزاگ

بیت اصلی

رابطه ریاضی

$2^n + 2^{n-1} + \dots + 2^1 + 2^0 = 2^{n+1} - 2^0$

در نهایت -2^0 کردن ←

مثال به روش Booth

شیفت و جمع

$$\begin{array}{r} 10010111 \\ \times\ 10111110 \\ \hline \end{array}$$

اولین یک بعد از صفر

بعد از یکی که به صفر می‌رسد 1 بگذار

به جای 1- بگذار

صفرهای بعد به صورت زیر می‌شود

$$\begin{array}{r} 10010111 \\ \times\ 10010111 \\ \hline 11\overset{0}{\cancel{1}}000010 \end{array}$$

$\rightarrow A \times (2^7 + 2^4 - 1)$

۳ جمع و شیفت داریم

شیفت و تفریق‌ها

مدار تقسیم کننده

الگوریتم ۲<

پردازش اعشاری چند تایی

اعداد

underflow

به توان ۲ بدهد ← توی ۸ بیت جا نمی‌شود

$x = 00000000.\boxed{00001001} \rightarrow x^2$ رخ underflow می‌دهد

$y = \boxed{10010000}.00000000 \rightarrow$

y^2 رخ over flow می‌دهد ← نمی‌تونه نمایش بدهیم

به توان ۲ که برسه توی ۸ بیت جا نمی‌شود ← over flow می‌کند

جمع و تفریق اعداد اعشاری چند شناور

نرمال شده‌ی مانتیس

$x = s \cdot 2^e \cdot (1+m)$

exponent

$x+y=?$

① تراز کردن توان‌ها

② جمع و تفریق کردن مانتیس‌ها

③ نرمال کردن مانتیس‌ها (بین ۱ و ۲ باید بشه)

$y = s' \cdot 2^{e'} \cdot (1+m')$

زیبا

$x+y = [s.(1+m) + s'(1+m')/2^{(e-e')}].2^{e} =$

$\boxed{s''.(1+m'').2^{e''}}$ ← نتیجه ممکن بود نرمال نباشد

$x.y = z \quad = (s \times s').2^{(e+e')}.[(1+m)\times(1+m')]$

ضرب مانتیس ها — جمع توان ها

ممکن است آتا ۲ باشد ← نرمال باید شود

اینجا جمع پیچیده‌تر از ضرب است.

توابع حسابی و مثلثاتی

رجوع به جدول — تایع $\boxed{\text{Table look up}}$ در فصل ۲

تخمین با استفاده از بسط‌ها

روش CORDIC

مثال: $1-x^2 = (1-x).(1+x)$

$x = 0.86 \longrightarrow 1-x^2 \equiv X$ معادل

$x = 1.80$

یا $\boxed{1-x^2 \equiv x}$ یک بار لازمی کند

CPU کمی فرض کن کار کمک پردازنده تمام شده و وقفه دریافت می‌کند

پردازنده کمکی اعداد اعشاری

عملیات اعشاری همچنان با اختلاف زمان بد است

CPU		co-processor
Data be processed	→	
	←	floating points maths
Answer returned		

CPU تمامی را به کمک پردازنده می‌دهد و صبر کار رسیدن را می‌کند هر موقع که کمک پردازنده جواب را پیدا کرد

کارهای اصلی لیستم و پردازش های بدیع

عملیات اعشاری را به یک کمک پردازنده دادیم

کارهای وقفه و عملیات floating point هالت عملیات به CPU

Session 20

Pipe line

پردازش خطوط لوله

روال اجرای نرم افزار به ترتیب و متوالی (لزومی الت)
برنامه‌های کامپیوتری (program) به صورت مجموعه‌ای از دستورات که در خانه‌های متوالی حافظه قرار دارند و اجرا می‌شوند

سیستم های چند هسته‌ای

امکان پردازش چند وظیفه به صورت همزمان

پردازش موازی

↑ میزان پردازش انجام شده در واحد زمان

↑ سرعت کلی پردازش در سیستم

↑ حجم و هزینه سخت افزار (افزایش)

قانون مور توضیح کند

طبقه بندی

سیستم های کامپیوتری امروزی به لحاظ پردازش موازی

طبقه بندی فلین

① SISD

Single instruction & Single data

توی این پردازنده ها در هر لحظه یک دستور و یک داده خوانده می‌شود و پردازش می‌شود، کاری که تمام شد میره دستور بعدی

مثل کامپیوتر پایه fetch, decode, execute

② SIMD

Single instruction multiple data

یک دستور را به چندین داده اجرا می‌کند

یک داده انجام بدی روی یک آرایه انجام بده (یا یک وکتور از داده)

vector processor

مثال: پردازش برداری مثل پردازنده‌ی گرافیکی

۱۹۲۵

۱۰۸۵ پرسش

Scanned by CamScanner

multiple instruction , single data : MISD

چند تا دستور متفاوت هم‌زمان روی یک داده انجام شود

مثال نداریم

ارتباط طبقه‌بندی فلین

multiple instruction, multiple data : MIMD

چند تا دستور و پردازش‌های مختلف روی داده‌های مختلف انجام بدهیم

multithread

مثال : پردازنده‌های چند هسته‌ای

کاربردی‌ترین نوع پردازش موازی

واحد کنترل هر هسته متفاوت است

حالا اتفاقاتی توی SISD افتاد :

خط لوله یا pipeline : تکنیکی برای افزایش سرعت در انجام یک سری دستورات کار متوالی - تکراری

آمادگی فرد توی تولید خودرو

مثال : خط موتاژ خودرو

کلید این کار : تفکیک کارهای تکراری

افزایش کارایی

کیفیت

۸۸

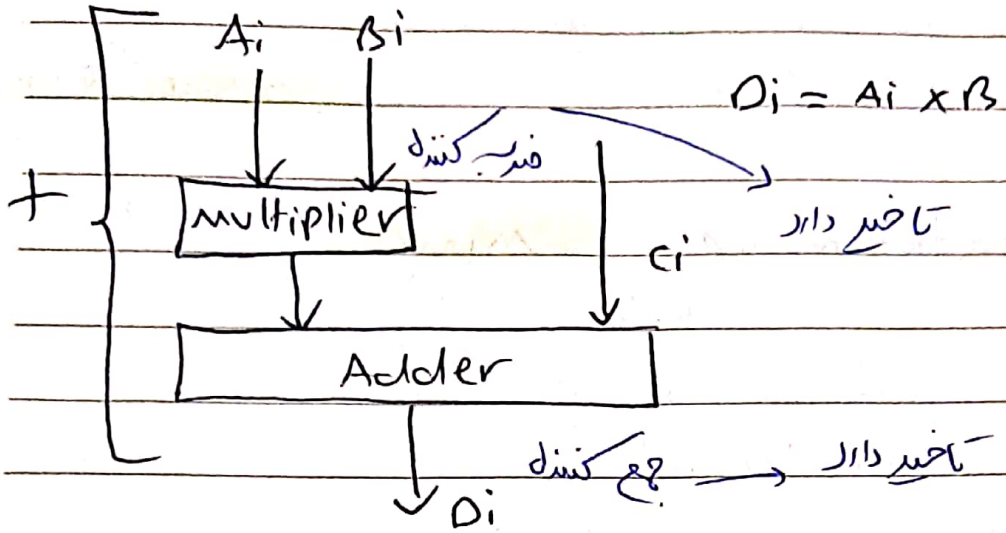
وقتی خودروی یک میلیارد بیرون بدافاصله بعد کی خودروی ۲ و ۳ ... هم میان بیرون ← برای تولید ماشین‌ها در انتهای روز بیشتر می‌شود تا اینکه چهار روز یک ماشین کار کنند حین overlap پیدا کردن کارها ← به این کار می‌گیم تکنیک خط لوله ← به کارگیری این تکنیک در سیستم‌های کامپیوتری برای ↑ کارایی سیستم

زیبا

توی کامپیوتر هم قرار است یک سری عملیات تکراری را پشت سر هم انجام دهیم

e.g. ex

$D_i = A_i \times B_i + C_i$



max ← تاخیر این دو واحد را باید بگیریم ← $\boxed{t_i}$ =

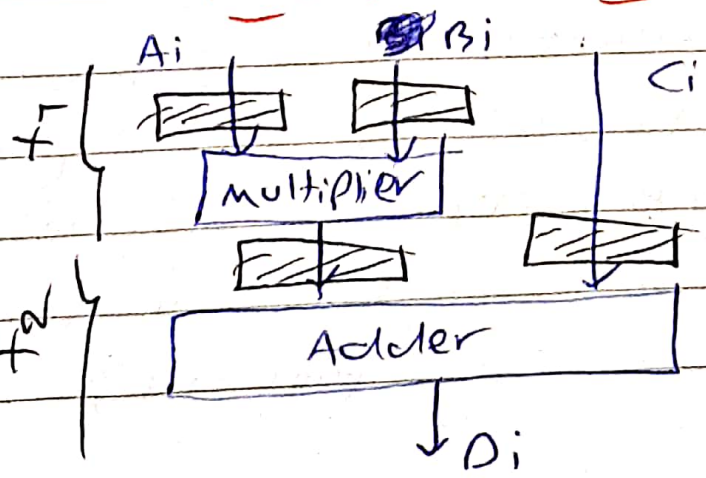
باید به این میزان صبر کنیم تا پردازش انجام شود تا مطمئن شویم جواب درست است

حالا خوبی می‌توانیم به کمک pipeline کردن را بالا ببریم ← تفکیک مراحل

بین بخش‌های مختلف، رجیستر می‌گذاریم

کار این رجیستر‌ها چه؟ اگر A_0 و B_0 تغییر کند تا زمانی که پالس ساعت به این رجیستر‌ها داده نشده، تغییر آن به واحد بعدی منتقل نمی‌شود ← این مراحل از هم تفکیک می‌شود

با این register ها یک pipeline می‌سازیم



$D_i = A_i \times B_i + C_i$

$T = (K + n - 1) \times t_p$

$t_p = \max\{t_1, t_2\}$

Scanned by CamScanner

بالای مدار

توضیح: در مرحله‌ی اول A_0 و B_0 را می‌بریم در ابتدای کار
بعد به اندازه‌ی یک تاخیر (t_1) باید صبر کنیم تا ضرب‌کننده کارش
را انجام دهد.

بعد از گذشت t_1 ، A_0B_0 و C_0 توی مدار ذخیره شده اند

لازم به جمع کردن می‌کند ← در این زمان multiplier می‌تواند بره سراغ
A_1 و B_1
تا ضرب کند

هم‌زمان با وقتی که A_0B_0 و C_0 دارن جمع می‌شوند
می‌تواند هم‌زمان ضرب A_1 و B_1 را حساب کند

یک مرحله overlap

برای n تا اعداد

تاخیر جمع‌کننده

$$T = (K + n - 1) \times t_p$$

تاخیر

تعداد مراحل

Pipeline کلاک

$$t_p = \max\{t_1, t_2\}$$

بعد اول K کلاک (به تعداد مراحل) باید بگیرد تا از خط لوله بیاد بیرون
بعد از اینکه اولین خروجی بیرون ← بیشتر کلاک دومی میاد بعد توی
برای باقی مانده‌ی عناصر ← فقط یک کلاک باید صبر کنیم

کلاک pipeline جمع‌کننده متناسب با کندترین کلاک باید باشد
$\max\{t_1, t_2\}$

speedup

میزان افزایش سرعت

بدون خط لوله

$$S = \frac{n \times t_i}{(K + n - 1) \times t_p}$$

$$\text{میزان افزایش سرعت} = \frac{\text{زمان اجرای سیستم کندتر}}{\text{سیستم کندتر}}$$

با خط لوله

$$S = \frac{n \times K \times \cancel{t_p}}{(K + n - 1) \times \cancel{t_p}}$$

فرض: خط لوله ایده‌آل باشد

اگر n زیاد شود
خیلی بزرگ بشه

$$\boxed{S = K}$$

Scanned by CamScanner

خط لوله ایده آل باشد یعنی ح ح وقتی یک کار بزرگی را به K مرحله می‌شکنیم

فرض می‌کنیم پالس ساعت به یکسان تقسیم می‌شود

$$\boxed{t_p = \frac{t_i}{K}}$$

$$\boxed{\text{زمان تاخیر مساوی}}$$ فرض

$$\rightarrow t_i = K \cdot t_p$$

از صفحه‌ی قبل ← $\boxed{S = K}$ ح یعنی هر چقدر pipeline را بشکنیم

سرعت به همان اندازه در حالت ایده آل

می‌تواند اضافه بشه

مسائل ح ① تا بی‌نهایت نمی‌توانیم این مراحل را بشکنیم

② فرض ایده آل (اینکه کلاک مساوی تقسیم شود)

③ مشکلات اجرا

clock per instruction

cpi _ pipeline برابر یک است یعنی تعداد کلاکی که برای هر دستور

می‌خورد یک است

مثلاً برای نوع مالتی ح یک مالتی تمام مراحل را طی می‌کند در یک کلاک

طول نمی‌کشد اما فرضی نمایی لیستم این طوریه که هر کلاک ح یک مالتی

آماده‌ی تحویل می‌شود ← ↑ کارایی

یک کل دستورات ← ① fetch ② decode ③ execute

register register

اگر یک رجیستر ولط این کارها بگذاریم

ح می‌تونه به یک خط لوله تبدیل کنیم ← یعنی نیازی نباشد صبر کنیم

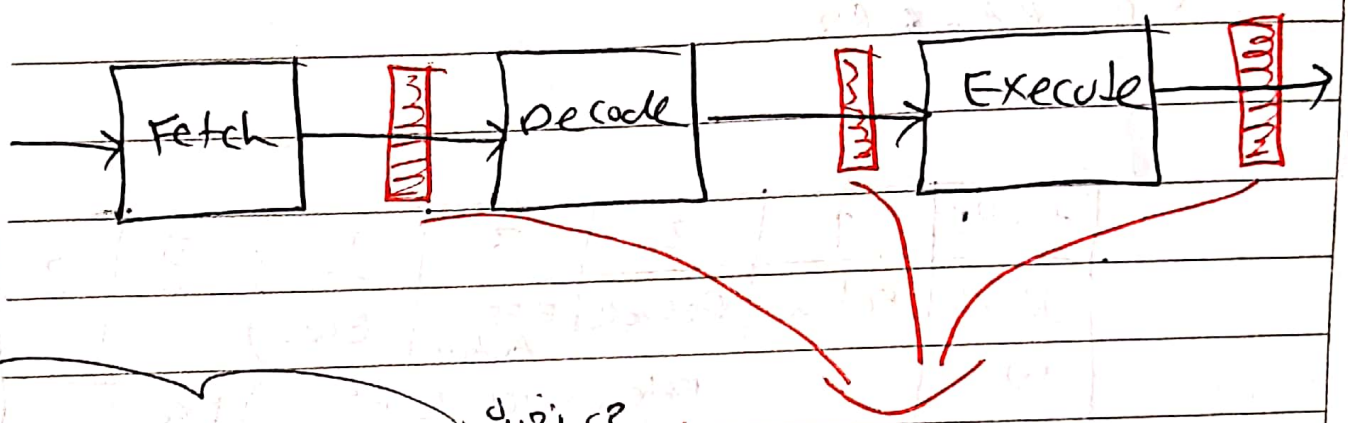
تا یک دستور خیلی بشه و دید بشه اگر بشه تا بعد بریم سراغ دستور

بعدی ح به جایی ح دستور اول را خیلی می‌کنیم در کلاک بعدی ح دستور اول دید می‌شود

دستور دوم فچ می‌شود و کلاک بعدی B دستور اول اگزیکیوت می‌شود و
دستور دوم دیکد می‌شود و دستور سوم فچ می‌شود ← اونی که خارج شد، پشت سرهم

دستور

اونی و اونی ← هم‌زمان بیرون ← توی SISD هم یک مقداری
توازی درست کردیم با overlap
کردن
↑ کارایی سیستم



کی ذخیره می‌کنه ← register

کلیه‌ی حالت برداری به ازای یک دستور

برای هر دستور چیزای مختلف ذخیره می‌شود تا دیتای دستورات با هم قاطی نشود ← ↑ پیچیدگی سخت‌افزار

اما سرانجام نیستیم ← چون منابع سخت‌افزاری داریم

همچنین کارایی خیلی هم نداریم بدون

در سیستم‌های امروزی pipeline هست ← ↑ کارایی

نمودار زمانی B

$\boxed{\text{اجرا} \quad 72}$

تلاش می‌کنیم توی یک سیستم تک‌هسته‌ای به $\boxed{CPI=1}$ برسیم

RISC ← مراحل اجرای دستورالعمل ها برابر و ترتیبی اند ← خیلی راحت می توانیم pipeline کنیم اما CISK ← چون مراحل دستورها فرق دارد ← نمی شه pipeline کرد

مخاطرات خطوط لوله
- وابستگی داده
- وابستگی کنترل
- منابع

شرایطی که در آن دستور جدید نمی تواند در پالس ساعت بعدی کارش را انجام دهد

وابستگی داده

```
ADD  R1,R2
MOV  R3, M[R1]
SUB  R4, R3
```

پالس ساعت	1	2	3	4	5	6
دستور اول	Fetch	Decode	EFF. Addr.	Exec.		
" دوم		Fetch	decode	Eff. Addr	Exec.	
" سوم			Fetch	Decode	Eff. Addr	Exec

stall

stall

دستورات یک کلاک تاخیر می خورند

تمام این مدت تا وقتی دستور دوم می خواهد آدرس موثر را بدست بیاورد تا دستور غیرمستقیم باشد M[R1]

R1 از دست قبلی بدست میاد و دستور قبلی داره execute می شه و هنوز تمام نشده و R1 حساب نشده تا دستور دوم بتواند آدرس موثر را از R1 بدست بیاره ← دستور دوم به خاطر وابستگی به داده ی اول ۲ گیر می کند

stall رخ می دهد → قرار بوده آدرس موثر کی تو کلاک ۴ حساب شه نمی تواند

یک کلاک تعطیل می کند ۲ یک کلاک تاخیر می خورد ← آدرس موثر در کلاک بعدی انجام می شود

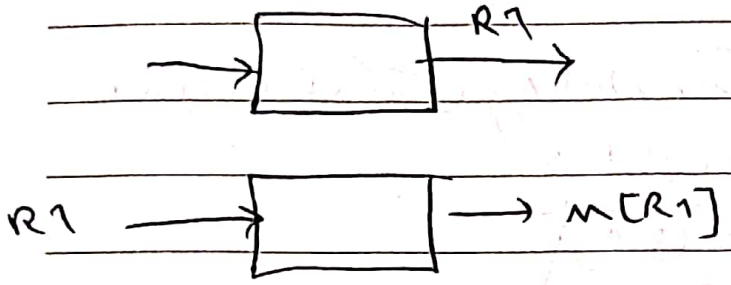
pipeline

تا وقتی دستور (د) از مرحله‌ی آدرس موثر خارج نشده و دستور بعدی هم
Stall می‌خورد ← هر چه دستور بعدی یک کلاک عقب می‌افتد.
هست

باید خیلی هوشمند باشد

از CPI=1 فاصله می‌گیریم

راه حل: کامپایلر بین دو تا دستوری که با هم وابستگی دارند،
یک دستور غیر وابسته قرار بدهد تا Stall نخورد



مشکل: توی یک کلاک
این دو تا با هم فعال
شده اند
هم زمانی عملیات این دو تا

تاخیر در اجرا ← CPI خراب می‌شه

راهکار:
اصلاح ترتیب عملیات در کامپایلر
تسریع در تحویل نتیجه در سخت افزار

وابستگی کنترل

```
SUB   R1,R2
BNE   L1
ADD   R3,5
JMP   L2
L1:   ADD   R3,9
L2:   ____
```

R1 = R1 - R2 ①
if (R1 == 0) ②
else
____ ③

در دستور اول: توی کلاک چهارم تازه می‌فهمد که نتیجه‌ی R1-R2 چی شده و
باید if را اجرا کرده یا else را

سخت افزار پیشرفته‌ی لازم برای افزایش کارایی

راه حل: کامپایلر

سخت افزاری ← به صورت داینامیک تصمیم‌گیری کنیم

احتمال

پیش بینی می‌کنیم که وقتی دلتورا اول خیلی شد ، احتمال قوی‌تر از اجرا می‌شه یا deadlock

خسارت ناشی از اشتباهات کنترلی > اشتباه داره‌ال

ممکن است چیزی تا دلتور را اشتباه آورده باشیم. و مجبور نیستیم دور بندازیم

راه حل وابستگی کنترل ← اصلاح ترتیب دلتورات در کامپایلر ✗ خوب نیست

اجرای دلتورات از هر دو مسیر شرط ← سخت افزار پیشرفته می‌خواهیم

پیش‌بینی نتیجه شرط

پیچیدگی

بهتر

وابستگی منابع

resource

$f_1 = f_5 \times f_3$

$f_{10} = f_6 \times f_7$

floating point

کلاک طول می‌کشه اجرا بشن

اگر دلتور بعدی هم floating point باشد

کامپایلر باید بین این دو تا دلتور ۶ تا دلتور مستقل بذاره که نیازی به واحد ضرب کننده‌ی floating نداشته باشند تا pipeline جلو محدودیت منابع گیر نکند

تقسیم از همه سخت‌تر ← ۲۸ تا کلاک طول می‌کشد

پس دو تا تقسیم floating باید ۲۸ تا کلاک فاصله بگیرند تا به مشکل نخورد پیشتم

قدار دادن دلتوراتی که به منابع وابسته نیستند بین دلتوراتی که به منابع وابسته اند

کامپایلر باید زحمت بکشد وابسته باشد!

Scanned by CamScanner

Session 27 — آخری

پردازش برداری

vector processor (پردازنده برداری)

SIMD ← single instruction multiple data

هدف: پردازش به صورت هزمان روی حجم قابل توجهی از داده‌ها

مثل پردازنده گرافیکی

CPI < 1 → می‌تواند کمتر از 1 هم بشود

چون چندتا دستور را همزمان انجام می‌دهیم

CPI —می‌شه→ IPC: instruction per cycle

هدف: IPC زیادتر کند تبدیل شه.

حداکثر IPC را یکی تر از 1 کنیم: ① هدر زیادی واحد اجرایی / واحد کنترل بین همه‌ی واحدهای اجرایی مشترک و واحد الت.

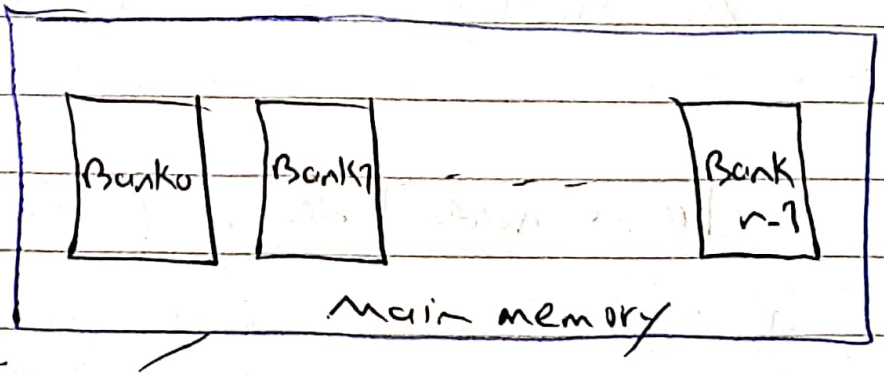
چالش اصلی: دریافت داده‌ها از حافظه ← با تاخیر بسیار زیادی انجام می‌شود

پهنای باند حافظه = گلوگاه اصلی

سیستم‌های پردازش موازی مثلاً 4 processor داریم که همشون می‌خواهند از یک حافظه‌ی کند، داده بگیرند

راه‌حل: تقسیم حافظه به بخش‌های کوچک

Memory interleaving



داده را می‌توانیم در یک زمان از بانک‌های مختلف حافظه بگیریم.

در هر دو مشکل حافظه هست

low order

high order

Interleaving

Low order memory Interleaving

شماره بانک

آدرس کلمه مورد نظر در بانک

Inter. Bank Addr.	Bank addr.

در این حالت ← کلمات متوالی یک آرایه در بانک های مختلف قرار می گیرند

عناصر متوالی به صورت همزمان قابل دسترسی خواهند بود

اگر یک بانک خراب شود ← کل آرایه دچار مشکل می شود

ضرر

اگر ۸ تا بانک داشته باشیم و ۱۶ تا دیتای پشت سر هم بخواهیم از هر بانکی ۲ تا خانه می گیریم ← ↑ اندازه

High order memory interleaving

در این حالت ← اگر یک بانک خراب شود، تنها کمتری از آرایه خراب می شود

کلمات متوالی یک آرایه در یک بانک قرار می گیرند

عناصر متوالی به صورت همزمان قابل دسترسی نخواهند بود

Bank Addr	Inter. Bank Addr.

در این حالت ← عناصر متوالی توی یک بانک پر می شوند وقتی این بانک پر شد می رویم سراغ بانک بعدی ← اگر یک تکه از یک بانک خراب شد، می رویم سراغ بانک بعدی

زیبا تر

Scanned by CamScanner

سیستم های چند پردازنده ای

فقط یک CPU نداریم چند تا داریم

از دو یا تعداد بیشتری هسته ی پردازشی به همراه حافظه و تجهیزات ورودی/خروجی تشکیل شده اند

می توانیم چندین هسته را روی یک تراشه قرار بدهیم (طبق قانون مور)

در شبکه ی MIMO ـ طبقه بندی فلین

چندراهه با چند دستور

Tightly coupled systems

ارتباط بین کامپیوترها از طریق شبکه کامپیوتری

هر کامپیوتر سیستم کامل عمل دارد

هر کامپیوتر وظیفه عملی دارد

این نوع سیستم ها از نوع سیستم های با اتصال سست محسوب می شوند

Loosely coupled systems

مهم ترین ضربه: ↑ کارایی سیستم به دلیل اجرای همزمان بخش های عملی برنامه نرم افزاری

↑ سرعت اجرا

مزایای چند پردازنده ای

کاربر می تواند برنامه را به بخش های غیر مرتبط تقسیم کند و هر بخش را به صورت همزمان با سایر بخش ها روی یک هسته ی مجزا اجرا کند

fault tolerance

مقاومت در برابر خرابی ها: کامپیوترها به تقسیم وظایف و توزیع آن روی هسته های مختلف (یکی خراب شود بقیه هستند) ↑ کارایی می بردازند.

ایده آل نیستند

باید برنامه نویس برنامه را مطابق سخت افزار بنویسد

Scanned by CamScanner

Kilo SOC ← روی یک تراشه، بیش از هزار هسته‌ی پردازشی هست

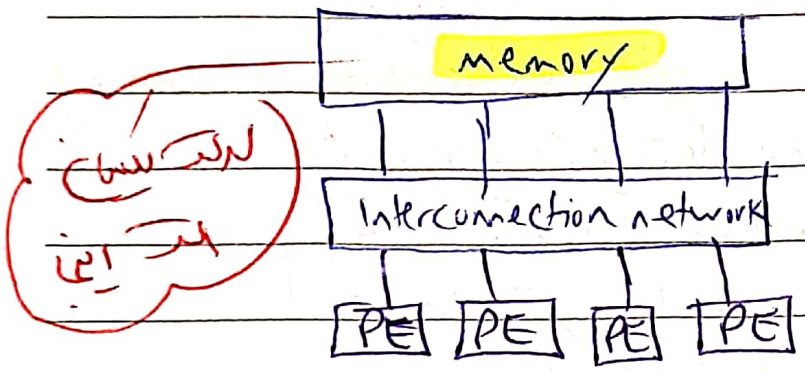
* در زمان ساخت تراشه ممکن است بخشی از تراشه خراب شود ← امکان fault

اما چون تعداد هسته‌ها زیاد است ← fault هم اگر رخ بدهد، باز هم در برابر

↑ امکان التقاره از سیستم ← خرابی مقاوم است

انواع سیستم‌های چند پردازنده‌ای

① سیستم با حافظه مشترک



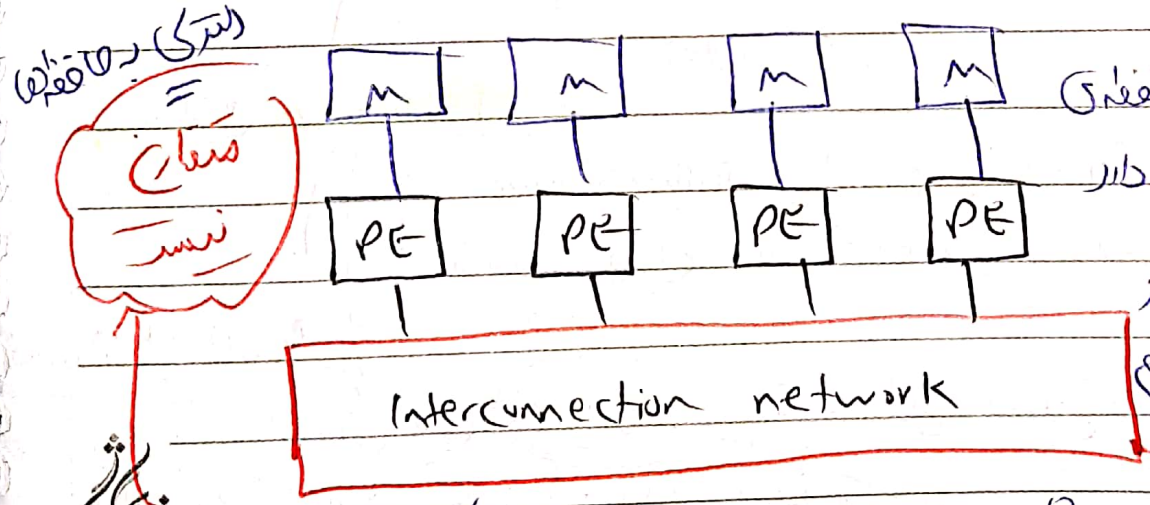
PE = processing Element

متقارن است

در کل سیستم این

یک حافظه داریم که بین همه‌ی PE ها مشترک است و PE ها به کمک یک ساختار ارتباطی به این حافظه متصل می‌شوند

② سیستم با حافظه توزیع شده:



دسترسی به حافظه‌ها = متقارن نیست

هر هسته حافظه‌ی محلی خودش را دارد

اما هر هسته می‌تواند به حافظه‌ی بقیه هم دسترسی داشته باشد

اما ✓ ① باید اجازه‌ش از PE مربوط به آن حافظه بگیرد

② یعنی کدهایی که اون داره‌ها کند می‌شه از اینه از حوزه‌ی خودش بردارد

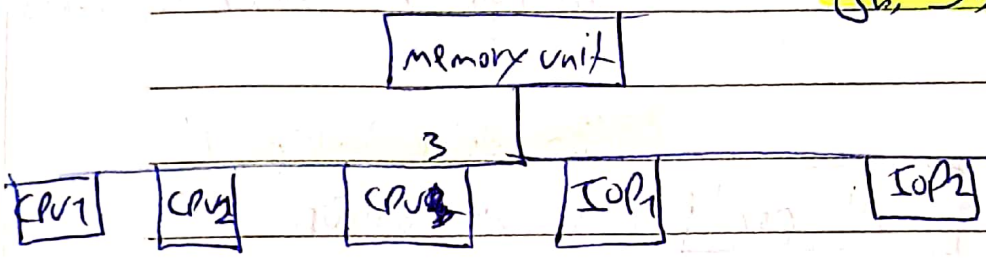
Scanned by CamScanner

ساختارهای ارتباطی

ساختارهای Interconnection network

- گذرگاه مشترک با ارتباط زمانی
- سیستم حافظه با درگاه چندگانه
- سوئیچ های crossbar
- سوئیچ چندمرحله ای
- hypercube اتصال

گذرگاه مشترک با اشتراک زمانی



بین CPU ها و memory و دستگاه های IO یک گذرگاه مشترک داریم

هر کسی با هر بخشی کار داشته باشد ← از این BUS باید استفاده کند

BUS مشترک در هر لحظه فقط می تواند داده صورت گیرد ← درخواست ها باید در طول زمان توزیع شوند

مثلاً می گیریم در ۳ ثانیه اول، درخواست A بده ← در ۳ ثانیه بعدی

روی محور زمان، درخواست ها قرار می گیرد

مزیت: سادگی، سخت افزار، ارزان بودن نسبی

عیب: کند بودن

از نظر performance خوب نیست

گذرگاه مشترک سلسله مراتبی (الزامات)

هر هسته پردازشی، باس عملی خودش را دارد تا به حوزه عملی خودش دسترسی داشته باشد

که می خواهد مربوط شود و برای ها صحبت کند چند تا پردازنده ← یک باس کلی می گذاریم

Scanned by CamScanner

system bus controllers

برای انتقال بایت های عملی به باس اصلی یک سیستم کنترلر گذاشتند تا کنترل کند
در هر لحظه فقط یک صبر و یک مقصد روی گذرگاه باشند.

ارزان است — مناسب ساختار سیستم چند پردازنده ای با حافظه ی توزیع شده
یعنی جایی خوب عمل می کند که بیشتر کار CPU با داده های عملی خودش است
باشد و فقط برای هماهنگی بخواهد با پردازنده های دیگر ارتباط برقرار کند

نقاط قوت
و ضعف ← (برعکس حالت قبلی است)

سیستم حافظه با دسترسی چندگانه

استفاده از حافظه های multi port ← حافظه ای که به صورت همزمان
بتوانیم چند کلمه را بخوانیم یا بنویسیم توسط CPU.

فایده: کارایی قابل قبول — رقابت پردازنده ها برای دسترسی به حافظه
ضرره: گران است

این ساختار برای تعداد هسته های کم مناسب است (نه برای تعداد بالا)

crossbar

سوییچ کلیدی بار

برقراری مسیرهای ارتباطی به شکل های مختلف

مشکل: اگر سوییچ ها بخواهند تعداد زیادی را پوشش دهند ← ممکن است
چند تا CPU بخواهند به MMU وصل شوند ← کل ترافیک میاد روی

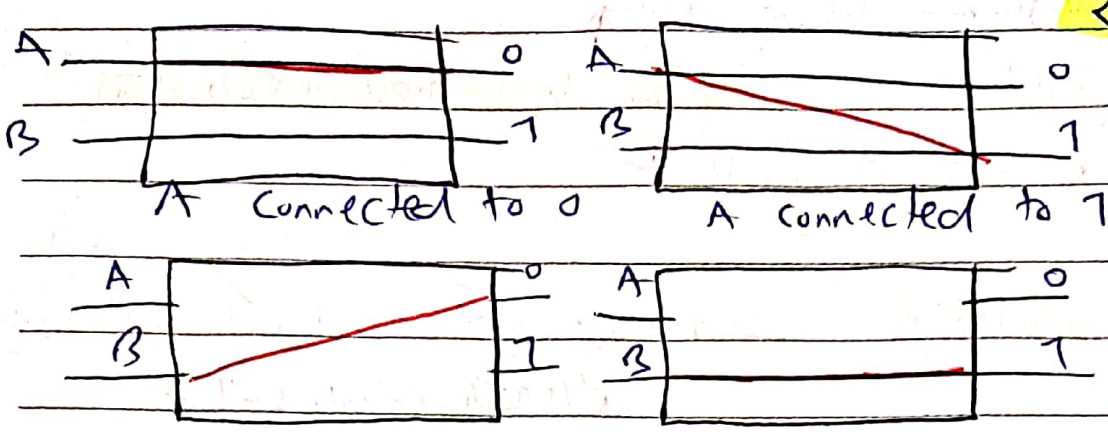
سوییچ ←

ایده ی سوییچ های چندطبقه (2x2)

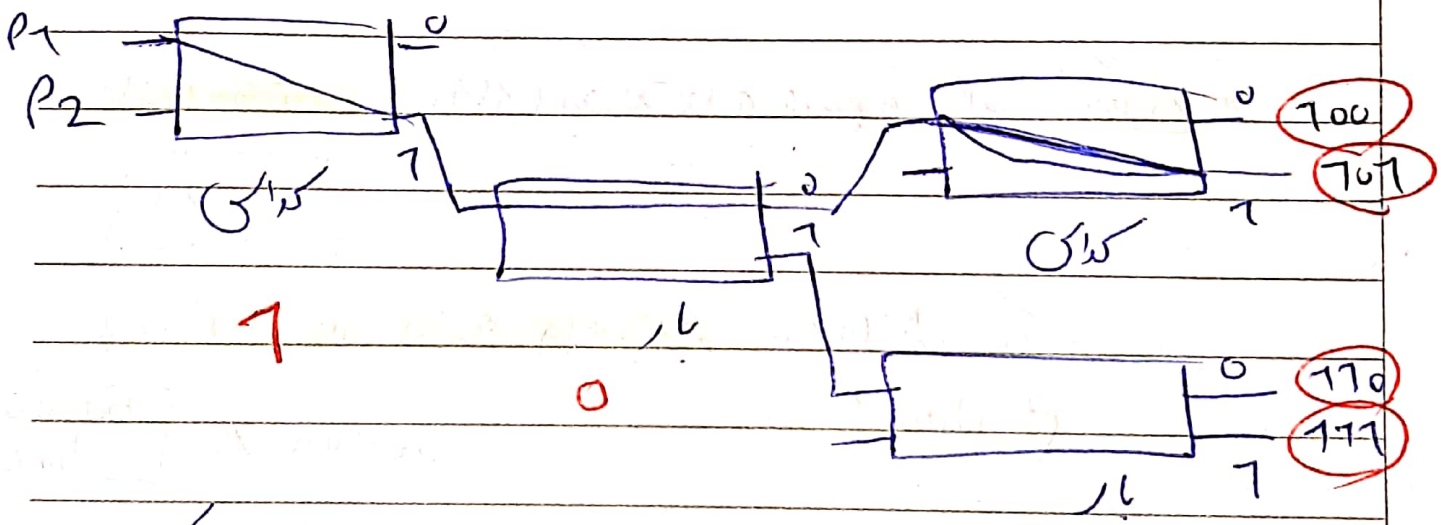
Scanned by CamScanner

Bar — Cross — crossbar

لوبیع ۲×۲



لوبیع چند طبقه — الدایه ۱۳ ← لوبیع ۲ به ۸ با ۳ طبقه لوبیع



دور ترین مسافت در شبکه

hyper cube اتصال

در طرح های ۲ بعدی ← طرح که بزرگ می شود، تعداد شبکه زیاد می شود

چینی اتصال به صورت ۳ بعدی ← حداکثر فاصله ای ۲ اتصال ۳

در گره باکره ی مجاور حداکثر یک بیت اختلاف دارد

Scanned by CamScanner

کنترل دسترسی به ساختار ارتباطی مشترک

ساختار زنجیره‌ای ← مثل ساختار توکن

اولویت هم زمانی

وقفه‌ای

ساختار موازی

توکن ← نوبت با کی است و چه کسی می‌تواند به باس دسترسی داشته باشد.

bus arbiter ← داور

ساختار زنجیره‌ای ← اولویت بر اساس مکان قرارگیری دستگاه است و اگر دستگاه چپ همیشه درخواست داشته باشد، نوبت به بقیه نمی‌رسد ← عدم انصاف در جایگاه گرانتی رخ می‌دهد.

الزامات ۱۶

ساختار موازی ← بر اساس درخواست‌ها و بازهم عدم انصاف گرانتی رخ دهد

الزامات ۱۷

جلوگیری از بروز هسته‌های مشترک ← حافظه مشترک

تبادل پیام

session 27 — 50 600 time

کنترل دسترسی به منابع مشترک ← ساختار پایه پیدا می‌کند

ناحیه بحرانی ← یکی از برنامه‌ها باید متغیر مشترک کار دارد

قفل کردن داده در ناحیه بحرانی را سمافور

اگر دو تا پردازنده، ناحیه‌ی بحرانی‌شون هم‌زمان به یک متغیر دسترسی پیدا کنند معلوم نیست نتیجه برنامه چی می‌شود.

باید جلوی یک کسی باشد تا در ناحیه‌ی بحرانی، فقط یک پردازنده به این متغیر دسترسی پیدا کند ← روش اول، پایه پیدا می‌کند

زیگزاگ

Scanned by CamScanner

رفتار باید پیروی به پردازنده master می‌آید بقیه پیرو می‌شوند

مشخص می‌کند که در هر لحظه کدام پردازنده‌ی slave حق دارد

یک متغیر دلتری پیدا کند ← تا وقتی کارش تمام نشده باشد

master به بقیه اجازه نمی‌دهد از این متغیر استفاده کنند

خب راه داریم اما تقارن لیستم را بهم می‌زند ← گفتیم تو لیستم ببینیم

کدوم master بیشتر وقتی بار بقیه را روی این می‌بندازیم

راه حل دوم: استفاده از تکنیک‌های نرم افزاری ← سخت افزاری

یک سری قفل که مثل متغیر هستند در حافظه تعریف می‌کنیم تا هر برنامه که

می‌خواهد وارد ناحیه‌ی بحرانی شود، منتظر بشه تا این قفل آزاد بشه

تا وقتی قفل است، گیر می‌کند، وقتی قفل آزاد شد ← وارد ناحیه‌ی بحرانی می‌شود

و خودش این متغیر را قفل می‌کند، کارش را انجام می‌دهد، وقتی خواست از ناحیه

بحرانی خارج بشه، قفل را آزاد می‌کند

گیر ممکن است دو تا پردازنده بیرون آمدن قفل را در یک زمان باز کنند

اینجا دیگر فرقی نداره کدوم پردازنده درخواست می‌دهد

```
while (lock);
lock = true;
```

→ در سخت افزار باید به مکانیزمی باشد تا این دو تا دستور بهم چسبیده شوند ← اگر جدا جدا اجرا شوند

⋮

lock = false

ممکن است یک پردازنده بیاید ولو و کار را خراب کند

به جزءهای کوچک شکسته نشود

اجرایش atomic

احتمالاً می‌گیم این دو تا دستور باید به صورت

حفظ سازگاری بین نسخه‌های مختلف یک داده

سیاست‌های به‌روزرسانی در حافظه‌ها

write through

write Back

Scanned by CamScanner