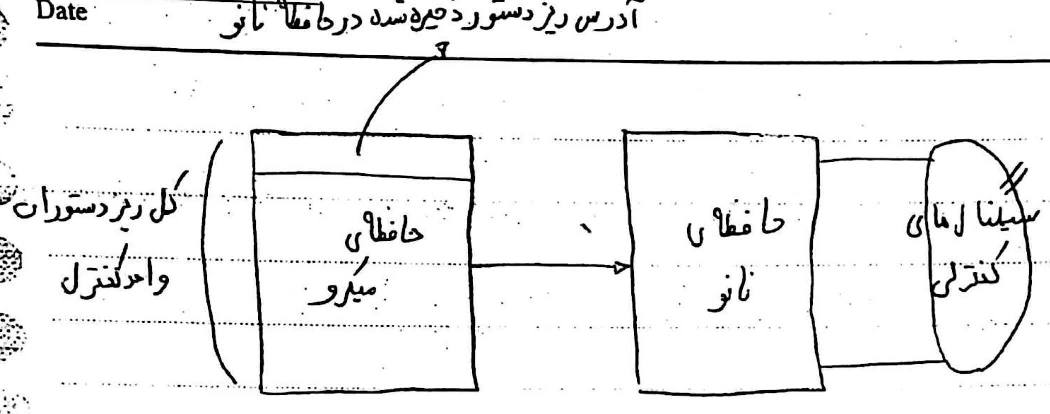
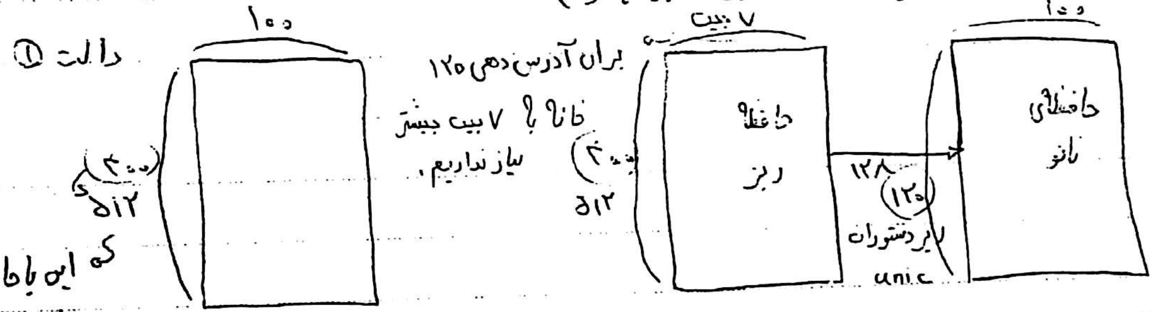


مثال) فرض کنید در یک سیستم ۱۰۰ سیگنال کنترل وجود دارد. اگر حافظه ریز حاوی ۴۰۰ ریز دستور باشد و از این ۴۰۰ ریزدستور فقط ۱۲۰ ترکیب متفاوت وجود دارد. در صورت استفاده از حافظه نانو چقدر در میزان حافظه صرفه جویی می‌شود؟ (۲۸۰ دستور تکراری مورد عجیبی نیست چون مثلاً برای دسترسی به حافظه چندین بار یک ریزدستور در کل برنامه اجرا می‌شود)



$400 \times 100 = 40000$ bit

$400 \times 7 + 120 \times 100 = 2800 + 12000$
$= 14800$ bit

⇐ پس حجم حافظه کم شد.

حافظه* (در واحد کنترل، قسمت های مختلف را اگر با حافظه پیاده سازی کنیم باید حتماً مضربی از ۲ باشد چون حجم آن مشخص است.)

* PLA ⇐ روش محاسباتی با PLA است. (یعنی مهم نیست مضرب ۲ باشد)

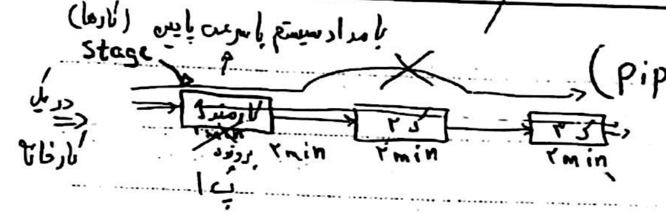
⇐ پیاده سازی multi cycle ⇐ یک سخت افزار را در دستورات مختلف به اشتراک گذاشتیم.

~~⇐ پیاده سازی پایپ لاین شروعش با single cycle است~~

Subject: با این روش همه کارها در همه زمان‌ها کار می‌کنند. — در این جا پایپ لاین خطی (نکات زیر راجع به خطی است)
Date

پیاده سازی پایپ لاین (خط لوله — pipeline)

با یک مثال:



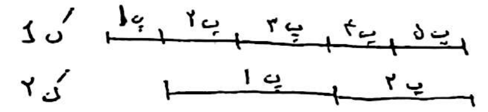
پ ۲   پ ۱

پ ۳   پ ۲   پ ۱ ⇐ ۳ تا پرونده در سیستم وجود دارد.

پ ۴   پ ۳   پ ۲

⇐ در MIPS ⇐ در پایپ لاین به طور هم زمان ۵ تا instruction داریم که به طور هم زمان در سیستم دارند اجرا می‌شوند.

stage های پایپ لاین از هم مستقل هستند ⇐ یعنی فقط خروجی های stage قبلی نگاه می‌کند ورودی آن پردازش می‌کند و ورودی stage بعدی را می‌سازد و کاری ندارد stage قبلی چه کاری (پروسس) روی داده انجام داده و فقط خروجی اش را می‌بیند.



⇐ کارها روی میز کارمند ۲ انباشته شد.

✓ همه‌ی stage ها با کندترین stage باید سنکرون (هم زمان) شوند. ⇐ با این کار ممکن است کارایی سیستم پایین آید ولی برای جلوگیری از پیچیدگی های طراحی این کار را می‌کنیم.

✓ هیچ stage ای قابل skip کردن نمی‌باشد.

پایپ لاین پردازنده MIPS:

stage ها:

(بستری با طول ۷ طبق تأخیرهای که قبلاً گفتیم)

① Istruction Fetch — IF — ۲ns

② Decode & Register Read — ID — ~~۱ns~~ ۲ns به دلیل قبل

③ Execution — EX — ALU ← ۲ns

Subject
Date

④ ALU → ۲ns MEM Memory Acces

⑤ WB write Back ~~۸ns~~ ۲ns به دلیل قبل

یک مثال)

lw R_1, ۱۰۰(R_0) ۸ns
lw R_2, ۲۰۰(R_0) ۸ns
lw R_3, ۳۰۰(R_0) ۸ns

$\Rightarrow 3 \times 8^{ns} = 24^{ns}$ در single

دستور

I_1: IF, ID, EX, M, WB — ۱۰ns
I_2: IF, ID, EX, M, WB — ۱۲ns
I_3: IF, ID, EX, M, WB — ۱۴ns

پایپ‌لاین: 14^{ns} و single = ۸ns
به نظر می‌رسد در اجرای یک دستور ⇐ پایپ‌لاین بدتر شود ولی در تعداد زیاد دستورات و overlap، Pipeline به مراتب بهتر است

$$speedup = \frac{24^{ns}}{14^{ns}} = 1.7$$

با فرض ۱۰۰ دستور:

single cycle: $100 \times 8ns = 800\,ns$
Pipeline : $10^{ns} + (100 - 1) \times 2^{ns} = 208^{ns}$
که 2^{ns} در هر مرحله اضافه می‌شود.

$$\Rightarrow speedup = \frac{800^{ns}}{208^{ns}} \simeq 3.9$$

که این افزایش سرعت یک حدی (کران) دارد.

در یک پردازنده k مرحله با زمان t موجود است. اگر n دستور وارد پردازنده شود میزان تسریع پایپ‌لاین در مقایسه با پیاده سازی تک مرحله‌ای چقدر است؟

single cycle: n.k.t
pipeline : $kt + (n-1)t$

حد بالای سرعت. (بی‌نهایت دستور)

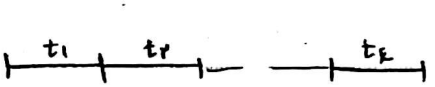
$$speed\,up = \frac{nkt}{kt + (n-1)t} = \frac{nk}{n+k-1} \xrightarrow{n \to \infty} k$$

که بنابراین مسئله ما مثلاً k هست می‌شود.

در حالت‌ها اره هم در واقع کمتر است چون ... هم می‌شود چون در single همان t نیستند و کمترند. (موارد کوچک‌تر)

Subject
Date

طالت کل زمان: $\sum_{i=1}^{k} t_i = T$



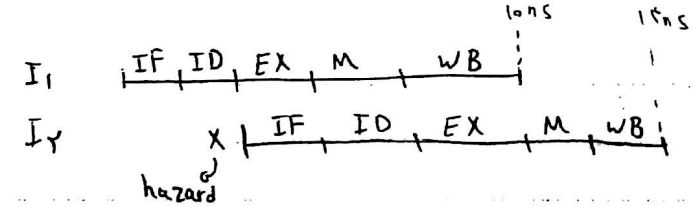
Single cycle : nT

pipeline : $T \leftarrow (n-1)\max(t_1, t_2, \dots, t_k)$ ، $k\max(t_1, \dots, t_k)$

$$\text{Speed up} = \frac{nT}{kT + (n-1)\max(t_1, t_2, \dots, t_k)} \xrightarrow{n \to \infty} \frac{T}{\max(t_1, t_2, \dots, t_k)}$$

این max وقتی رخ می دهد که همه ی t_1 تا t_k برابر باشند (چون جمع شان ثابت است) ⇐ ولی در MIPS این گونه نیست پس با ۵ هستا نمی رسیم.

گاهی اوقات برخی دلایل باعث می شوند دستور بعدی در سیکل بعدی وارد نشود و با این صورت پایپ لاین لنگ می شود که به آن hazard (مزاحم) می گویند ⇐ و speed up کم می شود.



درمثال قبل

⇐ اگر به هر دلیل نتوان در سیکل بعدی دستور جدیدی را وارد پایپ لاین کرد می گوییم پایپ لاین دچار مخاطره (hazard) شده است. ⇐ ما باید آن ها را بشناسیم و آن ها را رفع کنیم.

ما ۳ نوع hazard داریم: ۱- محدودیت سخت افزار Structure Hazard

۲- دستور دوم با نتیجه ی دستور اول نیاز دارد Data Hazard

```
add R1, R2, R3
sub R5, R1, R4
```

۳- وابستگی کنترل ⇐ مثلاً در حضور دستور beq Control hazard

ما ۲ راه حل برای این مشکل ها وجود دارد که ما یکی از آن ها را جدا کرده و بررسی می کنیم.

Subject
Date

۹۲/۸/۲۰:

* اگر با هر دلیلی نتوان در یک سیکل، دستور بعدی را وارد پایپ لاین کرد؛ می‌گوییم پایپ لاین دچار مخاطره Hazard شده است.

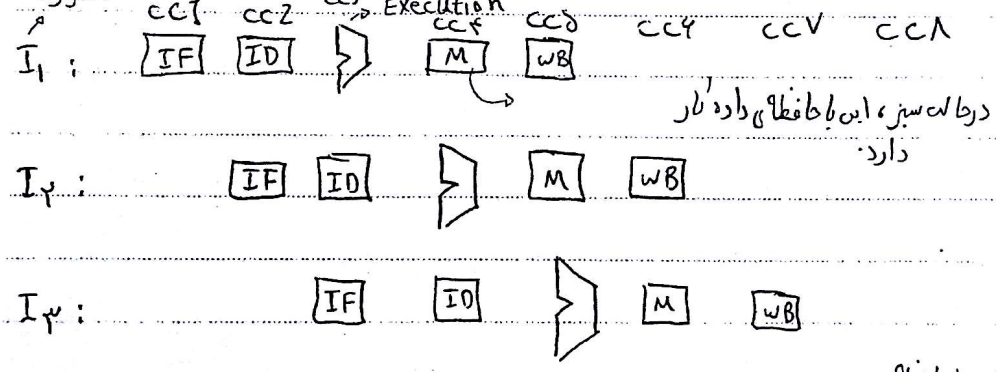
✍ انواع مخاطره‌ها:

① structural Hazard : محدودیت سخت‌افزاری

② Data Hazard : وابستگی داده‌ای بین دستورات

③ Control Hazard : " کنترلی بین دستورات

① فرض: در پردازنده‌ی MIPS یک طبقه‌ی نقش حافظه‌ی دستور و داده را ایفا می‌کند.



دلیل این مورد این است که در I های قبلی Memory access داریم و نمی‌توانیم Fetch را انجام دهیم.

⇒ پایپ لاین، با اندازه‌ی ۳ سیکل متوقف شده است.
stall

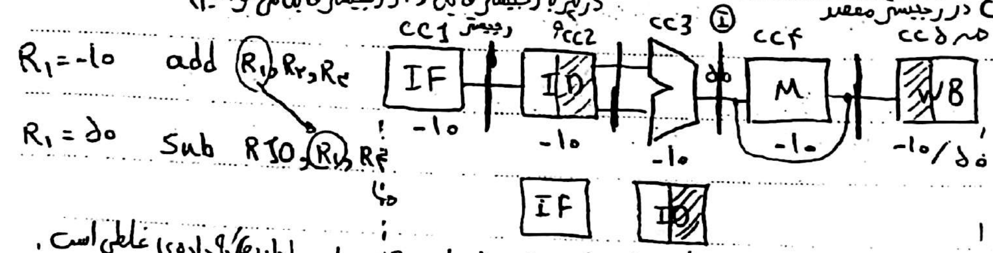
راه برطرف کردن: برداشتن محدودیت سخت‌افزاری

که از یک حافظه‌ی دستور و یک حافظه‌ی داده استفاده می‌کنیم.

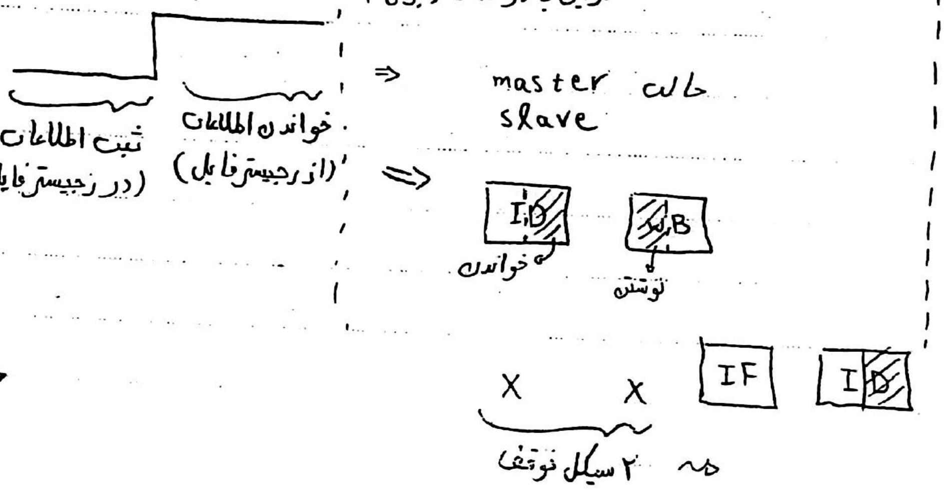
Subject

Date

۲) یک دستور با نتیجه‌ی حاصل از یک دستور دیگر که هنوز در پایپ‌لاین قرار دارد وابسته است.



که در این جا در sub (برای R_1، ۱۰- را داریم) که داده‌ی غلطی است.



که خیلی بد است

مراحل طراحی پایپ‌لاین: طراحی single cycle – بعد از هر stage، یک رجیستر قرار می‌دهیم (برای ذخیره‌سازی مثلاً fetch قبلی و عملاً IF را بی‌کار کنیم)

فرض کنیم دستور sub سر جای خودش وارد شود. (از رجیستر EX، ۵۰ را می‌آوریم)

sub:

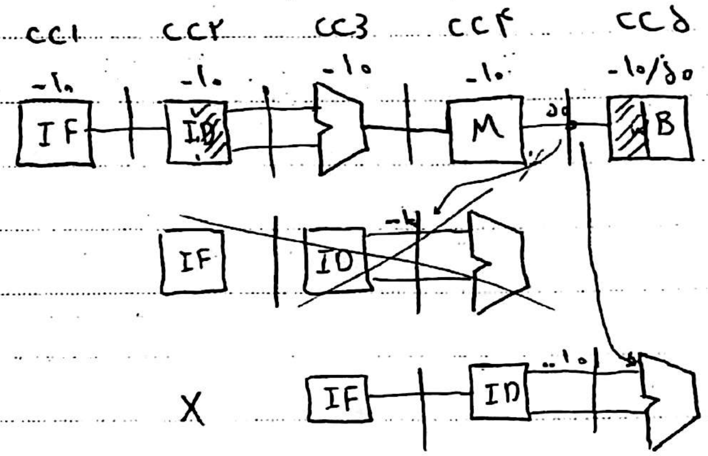
راه دوم: Data Forwarding می‌گویند ⇐ به این معنی که مثلاً ALU به جای این که از داده‌ای که در رجیستر فایل ثبت شده، از داده‌ای که بعداً در رجیستر فایل ثبت می‌شود استفاده می‌کند.

⇐ با کار بالا، توقفی در کار پردازنده نداریم. (که data forwarding را می‌توان با چند تا mux پیاده‌سازی کرد)

تیپ دستوراتی که در رجیستر فایل چیزی را می‌نویسیم ⇐ lw و R-type ها.

Subject
Date قبل از اجرای دستور lw
$R_1 = -10$

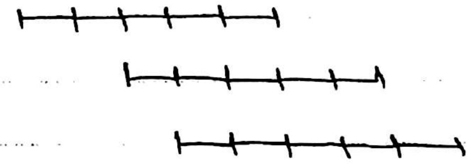
lw $R_1, 100(R\emptyset)$ — $R_1 = 50 \Rightarrow$ بعد از اجرای دستور lw
sub R_{10}, R_1, R_4



⇐ بنابراین برای R-type با data for. توانستیم مشکل را حل کنیم ولی در lw ناچار با توقف پایپ لاین هستیم و وابستگی را پذیرفتیم

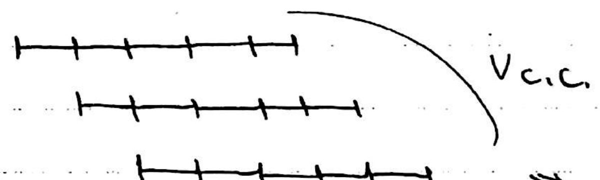
⇐ ما تا الان برای رفع hazard، سازمان طراحی پایپ لاین را تغییر دادیم ⇐ دیدیم این است که source مخاطره (وابستگی) را دستکاری کنیم:

lw $R_1, 100(R\emptyset)$
sub R_{10}, R_1, R_4
and R_5, R_6, R_7



8 c.c.

lw $R_1, 100(R\emptyset)$
and R_5, R_6, R_7
sub R_{10}, R_1, R_4



7 c.c.

⇐ scheduling یا زمان بندی دستورات.

که در این روش، در حالت static این را با کامپایلر انجام می‌دهیم که سخت است.

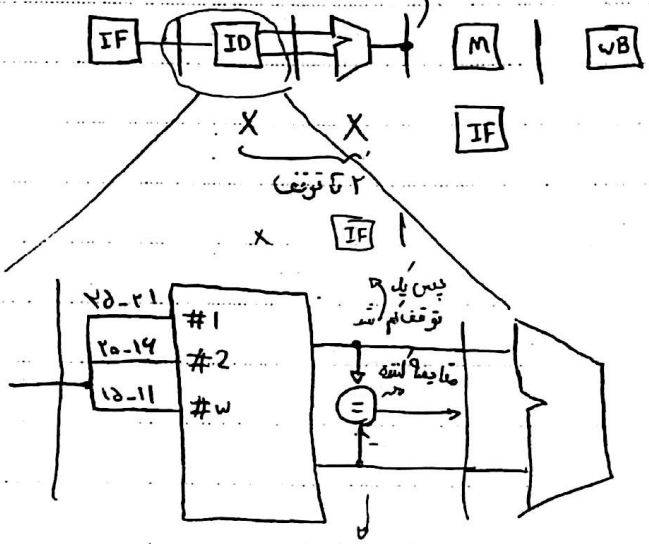
که در روش dynamic ⇐ خود پردازنده ترتیب اجرای دستورات را در نظر می‌گیرد و مثلاً ابتدا and و سپس sub را انجام می‌دهد ⇐ این مناسب‌تر است.

Subject
Date

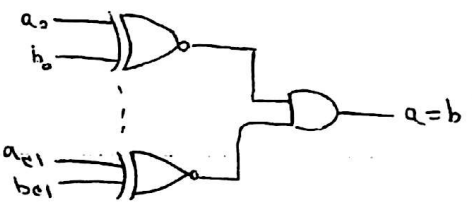
(۳) اگر یک دستور پرش وارد پایپ‌لاین شود، پیش از تشخیص برقراری / عدم برقراری شرط پرش نمی‌توان اقدام به fetch دستور بعدی کرد.

این جا مشخص می‌شود که add را انجام دهیم یا بپریم و sub نیم.

beq R_1, R_2, L_1
add R_{10}, R_{11}, R_{12}



L_1 : Sub R_{10}, R_{11}, R_{12}



⇐ مقایسه‌ی ۲ عدد ۳۲ بیتی (که این را criticalpath، ما را تغییر نمی‌دهد چون ما برای ALU، ۲ns تأخیر گرفتیم ولی این ۲ طبقه nand با مراتب تأخیر کمتری دارد.)

پس می‌آییم از Branch Prediction استفاده می‌کنیم: بدون توجه به وقوع شرط دستور بعدی را انجام می‌دهیم و در ۵۰٪ حالات درست عمل کرده‌ایم. (مثلاً دستور را اشتباه اشتباه گرفته بود دور می‌زند)

⇐ این ۵۰٪ - ۵۰٪ ⇐ همیشگی نیست و مثلاً در یک حلقه‌ی for مثل زیر، ۹۹٪ موارد اشتباه و فقط در مرحله‌ی آخر (۱٪) درست است پس این درصدها خود را تطبیق می‌دهد.

L_1 : ...

مثلاً ۱۰۰ بار
bne R_1, R_2, L_1 ⇒ اجرا می‌شود

جلسه‌ی بعد data path و کنترل پایپ‌لاین را توضیح می‌دهیم (در MIPS)

(۸۸) مبحث ترم اول ← تا آخر ارزیابی کارایی multi cycle نمی‌آید.

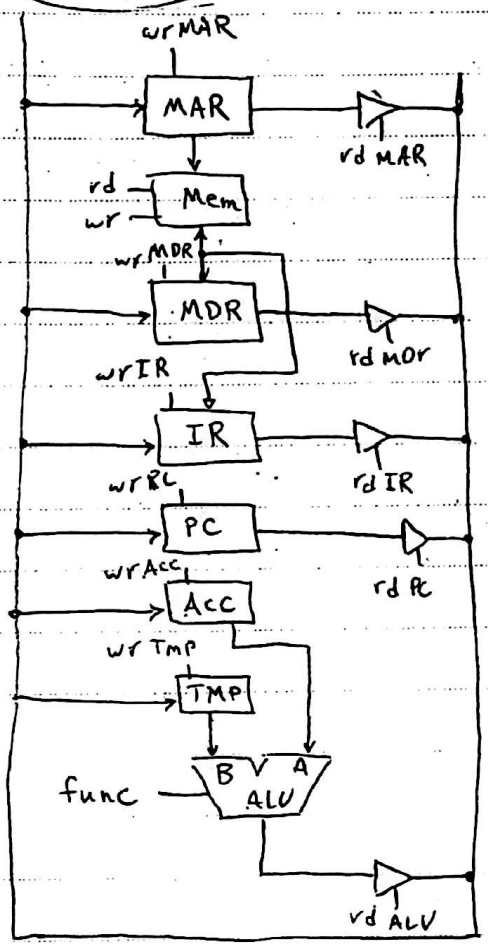
Subject

Date

که در قانون آمدال: مثلاً اگر کُفت ۵۰٪ دستورات تغییری نمی‌کند و موازی می‌شود ۷۰٪ با سرعت ۲ برابر و ۳۰٪ با سرعت ۵ برابر اجرا می‌شود رابطه‌ها اشکال در پاره می‌شود:

$$\frac{1}{\%50 + \frac{\%20}{2} + \frac{\%30}{5}} \Rightarrow$$

حل سؤال کوییز:



func	
000	A
001	B
010	A+B
011	A-B
100	B+1
101	A+1
110	A&B
111	NOT A

swap adr

M[adr] ⟷ Acc

fetch, ...

T_1: MAR ← PC ; TMP ← PC

T_2: IR ← M[MAR] ; PC ← TMP+1 ; بسیار مهم

T_3: Inst. Decode ; ← * همواره یک سیگنال برای decode می‌خواهیم *

T_4: MAR ← IR

T_5: MDR ← M[MAR] ; Tmp ← Acc ; این در T_4 قابل اجرا نبود چون bus اشغال شده بود.

T_6: Acc ← MDR ;

T_7: MDR ← Temp ; ⇒ این ۲ با هم قابل اجرا نیستند چون bus اشغال می‌شود.

T_8: M[MAR] ← MDR

⇒ سؤال پایان ترم ترم پیش

خیلی مهم

Subject
Date

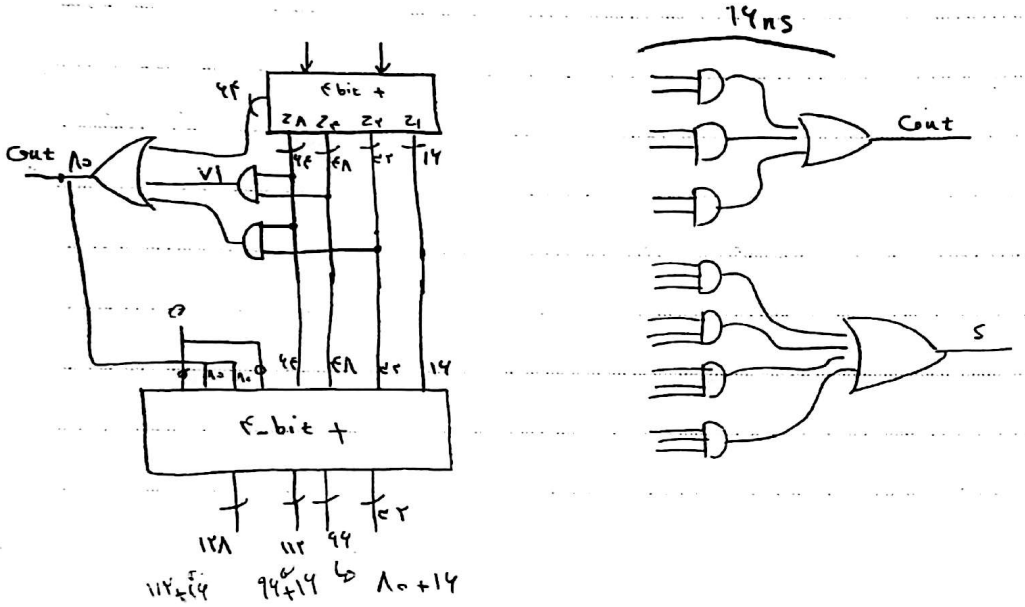
برای میان ترم، کد برنامه می‌توان سرجلسه آورد.

۹۲/۸/۲۷:

مثال میان ترم:

AND ۷ ns
OR ۹ ns ⇒ BCD تأخیر adder

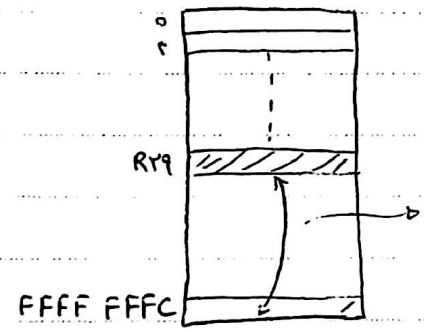
①



۱۱۲+۱۶ ، ۹۶+۱۶ و ۸۰+۱۶

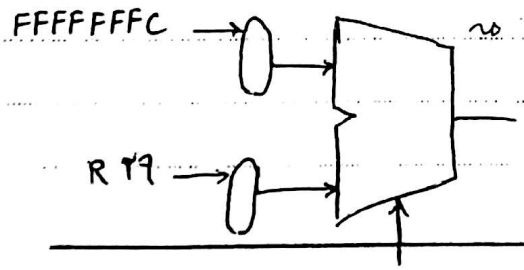
②

Stack_size R_i محتویات این رجیستر را با اندازه‌ی stack پر می‌کنیم (stack خالی)



اختلاف این ۲ رام خواهیم

معادل با: sub FFFFFFFC_R۲۹



(۹۵) استاندارد IEEE :

محدوده توان را متقارن انتخاب می‌کنیم $-43 \le E \le 43$ ⟸ $0 \le E \le +86$ محدوده نمایش مقدار

Subject

Date $0000000 \le E \le 1111111$ Biased −۴۲

برای توان عدد ۰

(۳) بزرگترین و کوچکترین عدد قابل نمایش با این را مشخص کنید، از روش implicit one استفاده کنید

S (1)	E (7)	M (12)

$-2^6 \le E \le 2^6-1 \Rightarrow -64 \le E \le 63$

کوچکترین عدد مثبت : برای حالت خاص

$+(1+0/99\ldots9)*2^{99\ldots9 / -42}$

این نمایش عدد صفر است که کوچکترین ⟸ 0000000 عدد مثبت

کوچکترین عدد مثبت $+(1+0/00\ldots01) \times 2^{0000\ldots0}$ (2^{-12})

یا $(1+2^{-12})*2^{-43}$

بزرگترین عدد مثبت : $+(1/1\ldots1)*2^{+43} = (2-2^{-12})*2^{43}$

---

(۴) یک پردازنده‌ای داریم که ۴ نوع دستور ALU و load ، store ، Branch داریم ؛

		clock cycle C.C.
ALU	۴۰٪	۱
Load	۲۵٪	۲
Store	۱۵٪	۲
Branch	۲۰٪	۲

اکثر دستورات lw ها که توأم با اسعمال

که می‌خواهیم کار را در یک $lw\ R_1, 100(R_2)$

تا از حافظه انجام دهیم (جمع) $add\ R10, R10, R1$

$\Rightarrow R10 \leftarrow R10 + M[100+R_2]$

با جای دستور ها می‌خواهیم دستور مستقیم زیر را اجرا کنیم. $addM\ R10, 100(R_2)$ C.C ۳

⟸ برای اجرای دستور ها تغییراتی در مسیرداده و واحد کنترل داده ایم که C.C مربوط به Branch را ۲ به ۳ تبدیل کرده است.

⟸ اگر ۱۵٪ از دستورات ALU از این نوع باشد آیا اضافه کردن این دستور با صرفه هست یا خیر؟

حل اول درحالت : $CPI_1 = 0/4 \times 1^{c.c.} + 0/25 \times 2^{c.c.} + 0/15 \times 2^{c.c.} + 0/2 \times 2^{c.c.} = 1/6^{c.c.}$

$0/4 \times 0/15 = 0/06$ ۱۵٪ از ۴۰٪ ⟶ ۶٪ از دستورات کل ما

$CPI_2 = [(0/4-0/06)*1^{c.c.} + (0/25-0/06)*2^{c.c.} + 0/15*2^{c.c.} + 0/2\times 3^{c.c.} + 0/06*3^{c.c.}] \times \frac{100}{94}$

⟵ یا همان تقسیم بر $\frac{94}{100}$ ⟸ طبق مطالب صفحه‌ی بعد

Subject
Date

۱۰۰ ← تعداد دستورات قبلاً ما
- ۴۰ − ۴
- ۲۵ − ۲
- ۱۵
- ۲۰
- +۹۶ ⇐ پس تعداد دستورات ما در این حالت ۹۴ شده است.

⇐ هر کدام CPI آن بیشتر شد بدتر است یعنی اگر $CPI_2 < CPI_1$، بنابراین اضافه کردن این دستورها مفید است.

---

⑤ اگر این ۲ برابر باشند دستور بعدی را بی‌خیال شو. R_i , R_j ⇐ skip_next بدرد else , if می‌خورد.

⇐ می‌خواهیم مسیر داده و واحد کنترل آن را در پردازنده‌ی MIPS ایجاد کنیم.

پس مسیر داده‌ی آن خود به خود در پردازنده‌ی ما وجود دارد. ⇐ beq R_i , R_j , 1 ← دستور بالا دقیقاً این کار را می‌کند و معادل است با

⇐ استفاده از این دستور:

```
    skip_next  Ri , Rj
    jmp        L1
L2:
    ⋮
L1:
```

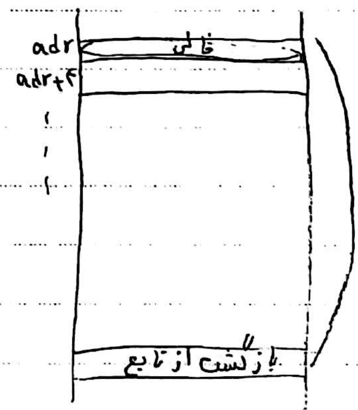
---

⑥ روش دیگری برای فراخوانی تابع وجود دارد که به صورت روبه‌رو است: csr adr

این دستور pc+4 را در آدرس adr ذخیره کرده و pc را برابر adr+4 قرار می‌دهد.

مسیر داده‌ی آن را می‌خواهیم رسم کنیم.

۳۱ ۲۶ ۲۵	۰
csr	adr



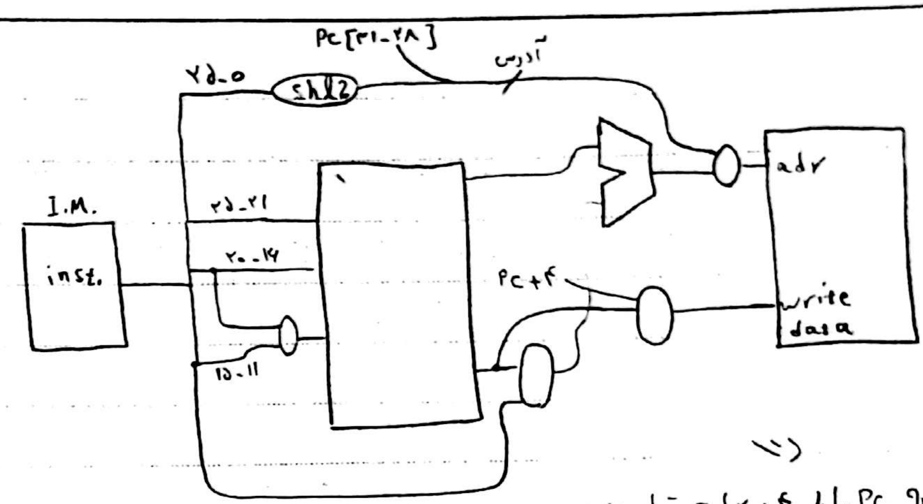
⇐ در نوشتن تابع، همیشه خانه‌ی اول تابع را خالی می‌کنیم (رها می‌گذاریم) که و پس از فراخوانی، آدرس برگشت (pc+4) را در خانه‌ی اول می‌نویسد.

⇐ در این فرم، صدا زدن تابع دیگر در تابع مشکلی ندارد ولی ایراد آن این است که اگر در خود تابع

یا تابع بعدی، همین تابع را صدا کنیم به مشکل می‌خوریم و آدرس بازگشت را از دست می‌دهیم.

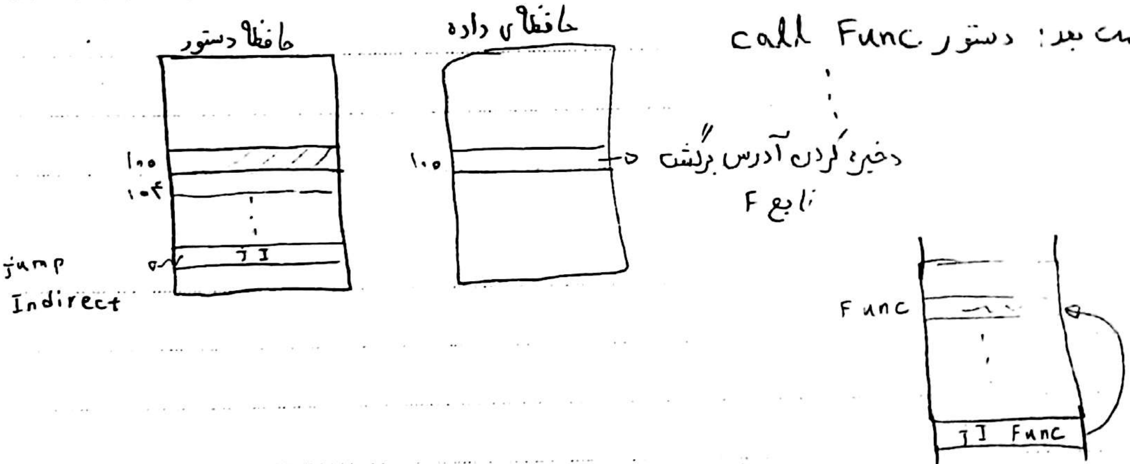
Subject
Date

ما چون در حافظهٔ دستور نمی توانیم بنویسیم، آن را در خانهٔ متناظر در حافظهٔ داده می نویسیم.



برای این که PC را با adr+۴ قرار دهیم، هم باید adr ساخته شده را با ۴ جمع کنیم و هم با یک mux آن را با PC برگردانیم که در بالا نشان داده شده است.

قسمت بعد: دستور call Func



۷ دقت سیستم نمایش عدد ما چیست؟ کوچکترین فاصلهٔ بین دو عدد متوالی در کوچکترین توان

اولین عدد ما ⇐ ∅ _ . _

دومین عدد ما ⇐ ○ _ _ _

کم می کنیم

دقت نمایش

∅ 0.000 0000

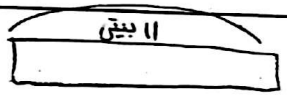
1.00000×2^{-7}

1.0001×2^{-7}

0.0001×2^{-7}

$2^{-4} \times 2^{-7} = 2^{-11} \Rightarrow$ دقت ما

Subject
Date

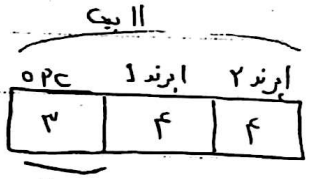


مثال بعدی :

۳ نوع دستور داریم : ۲ اپرندی ؟
۱ اپرندی ۴۰
۰ اپرندی ۱۲۸

فیلد آدرس اپرند ۴ بیتی است.

حل :

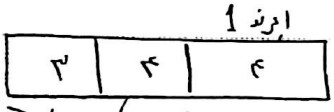


اگر فقط ۲ اپرندی داشتیم $۸ = ۲^۳$ تا دستور خواهیم داشت ⇐
ولی این ۸ تا کُدها همان ۳ تا opcode و ۸ حالت آن را با ۲ اپرندی اختصاص نده.

$۲^۳ - ۴$ باقی مانده

۴ : تعداد دستورات ۲ اپرندی

مرحله ۱ و مرحله ۲ از بیشترین شروع کردیم و پایین آمدیم.



پس وقتی ۴۰ تا را کم کردیم بقیه می‌تواند با ۰ اپرندی اختصاص یابد.

$(۲^۳ - ۴) \times ۲^۴ = ۴۰$ ⇐ دستورات باقیمانده برای ۱ اپرندی

تعداد دستورات ۱ اپرندی

تعداد دستورات ۰ اپرندی ⇐ $[(۲^۴ - ۴) \times ۲^۴ - ۴۰] * ۲^۴ = ۱۲۸$

⇐ در این مسیر حل هر کدام از موارد می‌تواند مجهول ما باشد مثل ۴۰، ۱۲۸ و ... .

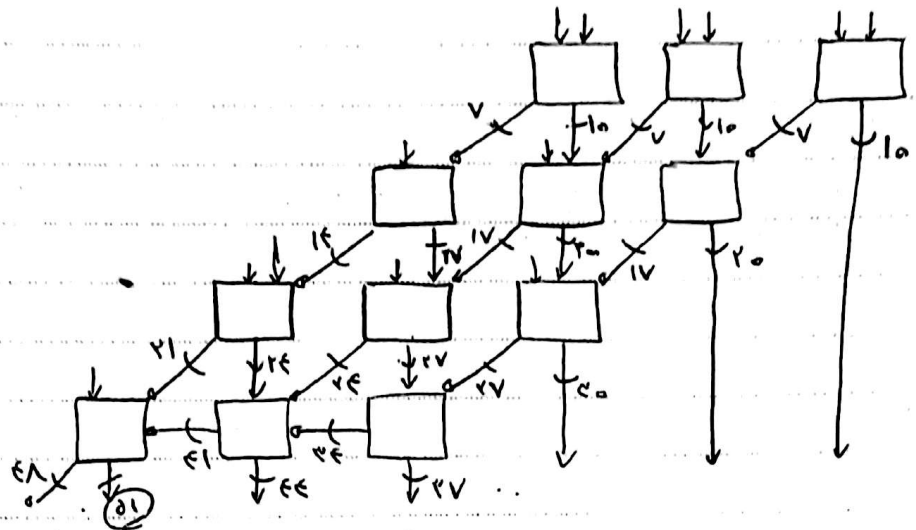
مثال) فرض کنید ۸ بیتی ، تأخیرش را خواستا (با تأخیراتی که در شکل نشان داده‌ایم) ؟

"ما برای ۴ بیتی حل می‌کنیم و ۸ بیتی هم مشابه است."

حل در من بعد

Subject
Date

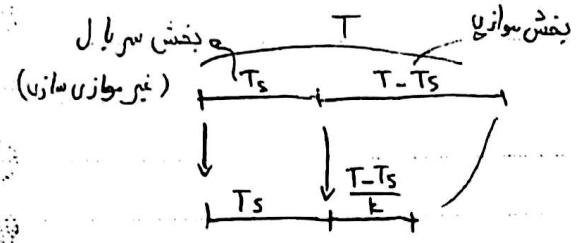
امتحان رأس ساعت ۷:۱۵ آغاز می شود



می توان فرمولی براساس بیشترین تأخیر جمع یا بیشترین تأخیر ⇒ به دست آورد Count

---

مثال بعدی : قانون آمدال :



یعنی مثلاً سرعت add، ۲ برابر شود.

تعداد پردازنده ها : (k)

T_s : بخش سریال برنامه (غیرقابل موازی سازی)

مثلاً : ۱۰٪ برنامه ترتیبی است
۱۰ پردازنده

$$\text{speed up} = \frac{T}{T_s + \frac{T - T_s}{k}}$$

$k = \infty$ یعنی

$$\text{speed up} = \frac{1}{\frac{0.9}{10} + 0.1} = \frac{1}{0.09 + 0.1} \simeq 5.1$$

حد تسریع ما $\left(\frac{T}{T_s}\right)$ است (در مثال ما ۱۰ است) ⇐ پس ما نمی آییم تعداد پردازنده را اکلی بالا ببریم چون از یک جایی به بعد تأخیر ترم دوم در مخرج کم می شود. پس باید توجیهی برای تعداد پردازنده ها داشته باشیم.

در آزمایش بعدی، نظر شما را بیان کردند که هرچه تعداد پردازنده ها بیشتر باشد سرعت افزایش می باید ولی آمدال می گوید این گونه نیست و دلیل آن این است که آمدال با واقعیت مطابقت ندارد چون در واقعیت T هم با افزایش پردازنده ها افزایش می یابد ولی آمدال آن را در نظر نمی گیرد.

Date: Subject:

۹۲/۹/۹:

۲ تا راه‌کار برای مقابله با Hazard در پایپ‌لاین:

① حفظ وابستگی و تغییر سازمان پایپ‌لاین

② تغییر ترتیب دستورات برای از بین بردن وابستگی موجود در برنامه

② ـ ۱ زمان‌بندی ایستا ⇐ static scheduling ⇐ سورت آف‌لاین

② ـ ۲ " پویا ⇐ Dynamic ⇐ سورت آنلاین (در حین اجرای برنامه)

حفظ ۲ ویژگی اساسی برنامه:

① جریان داده در برنامه

② رفتار استثنایی Exception Behavior

اگر یک دستور پیش از تغییر ترتیب دستورات باعث بروز Excep. شده است پس از تغییر ترتیب دستورات باز هم باید باعث بروز Exep شود و برعکس.

Exep دارد. ⇐ پس این اشتباه است.

(مثال جلوگیری از دسترسی به آدرس ∅)

```
beq R1, R0, L1
lw  R10, 0(R1)
```

```
R1 = 0
lw  R10, 0(R1)
beq R1, R0, L1
L1:
```

پردازنده با این نتیجا می‌رسد Exep ندارد.

آیا با این کار Performance بالا می‌رود؟

∅ ⇓ مثلاً آدرس ∅ مثلاً آخرین است.

L1:

Raz

Date: Subject:

حال به بررسی جریان داده می پردازیم:

فرض: دستور i مقدم بر دستور j است.

i
⋮
j ← به ترتیب در برنامه عادی

انواع وابستگی ها:

جاها باید با سخت افزار این کار را انجام داد. حرکت کرد.

که این موارد را در برنامه نمی توان کاری کرد و نباید برای افزایش کارایی دستورات را جابجا

① Data Dependence : دستور i در رجیستری می نویسد که دستور j از آن رجیستر می خواند یک جریان داده داریم.

i add R_1, R_2, R_3 → Read After write Hazard (RAW)

j sub R_{10}, R_1, R_5

نمی توان sub را جابجا قبل از add منتقل کرد.

② Name Dependence : در این جا در واقع جریان داده ای وجود ندارد ولی با عملیات rename (مثلاً) ، مشکل ایجاد می کند.

۱-۲ Anti - Dependence یا دوابستگی.

دستور i از رجیستری می خواند که دستور j در آن رجیستر می نویسد. در این صورت اگر دستور j پیش از دستور i اجرا شود، دستور i به اشتباه مقدار جدید این رجیستر را می خواند.

{ i add R_1, R_2, R_3
{ j sub R_2, R_5, R_4 → R_{10}

→ اگر sub را جابجا قبل از add برای افزایش Performance ببریم مشکل آفرین است ولی جریان داده ای نداریم فقط تشابه نام است.

{ j sub R_{10}, R_5, R_4 → write After Read Hazard (WAR)
{ i add R_1, R_2, R_3

این مشکل با Register Renaming قابل رفع است و دستورات را می توان جابجا نمود برای بهبود کارایی.

Date: Subject:

۲-۲) وابستگی خروجی: output Dependence

دستور i در رجیستری می‌نویسد که دستور j نیز در همان رجیستر می‌نویسد در این صورت اگر دستور j پیش از دستور i اجرا شود، مقدار نهایی این رجیستر به اشتباه توسط دستور i تعیین می‌شود.

```
i   add  R1 , R2 , R4
j   sub  (R1) , R5 , R6        WAW Hazard
```

در انتها نتیجه‌ی sub را دارد.

```
j   sub  R1 , R5 , R6    => Register Renaming
i   add  R1 , R2 , R4
```

این هم با Register Renaming قابل رفع است.

در انتها به اشتباه نتیجه‌ی add را دارد.

دستورات floating point ⇐ سیکل اجرای آن‌ها طولانی است و بنابراین نگه داشتن پایپ‌لاین خیلی طولانی می‌شود ⇐ پس اگر بتوانیم این مشکل را رفع کنیم خیلی در زمان صرفه‌جویی کرده‌ایم.

مثلاً div.d F0 , F2 , F4 ← وابستگی داده

mul.d F10 , F0 , F8

۱۷ سیکل طول می‌کشد و به ALU نیاز دارد.

مثال)

```
for(i = 999 ; i != 0 ; i--)
    x[i] = x[i] + s ;
```

به تمام عناصر یک آرایه ۱۰۰۰ عنصری از اعداد ممیز شناور با دقت مضاعف یک عدد ثابت را اضافه می‌کند.

وابستگی داده‌ای بین دستورات اول و دوم

دستور اول	دستور دوم	تعداد سیکل‌های توقف (همان stall)
FP ALU	FP ALU	۳
〃	SW	۲
LW	FP ALU	۱
〃	SW	0

Raz

کار مفید ⇐ خواندن محاسبه نوشتن ایده آل ۳ سیکل

Date: Subject:

```
Loop: lw.d    F0, 0(R1)
      add.d   F4, F0, F2      متوقف کردن
      sw.d    F4, 0(R1)
      addi    R1, R1, -8      بایت
      bne     R1, R0, Loop
```

برنامه عادی (بدون حلقه) : ۹ سیکل طول می کشد.

⇐ برای پیاده سازی حلقه نیاز به سربارهایی داریم :

① update of loop variables

② چک کردن شرط حلقه

پس این ها سیکل را بالا می برند — دستورات سربار

این به خاطر این است که در دستور بعدی روی R_1 می خواهیم تصمیم گیری کنیم پس یک سیکل توقف می کنیم.

```
→ Loop: ① lw.d   F0, 0(R1)
        ② addi   R1, R1, -8
        ③ add.d  F4, F0, F2
        ⑥ sw.d   F4, 8(R1)
        ⑦ bne    R1, R0, Loop
```

این برنامه ۷ سیکل طول می کشد که باز هم نسبت به حالت ایده آل خیلی دور است.

static sch. : ۷ سیکل

به خاطر این است که یکبار R_1، ۸- شده و برای اینکه به جای اشتباهی نرود.

→ for (i = 999; i != 0, i -= 2) {

x[i] = x[i] + s;

x[i-1] = x[i-1] + s; ⇐ پس در این جا به جای پذیرفتن ۱۰۰۰ بار سربار، ۵۰۰ بار سربار را می پذیرد (i -= 4) ⇐ ۲۵۰ بار می پذیرد

}

در حالت max این است که کل حلقه را باز کنیم و ۱۰۰۰ تا دستور بدون سربار بنویسیم. ⇐ loop unrolling

باز کردن حلقه

(باز کردن حلقه برای این مثال درس بعد)

Raz

قبل ها برای یک کار مفید بودند

Date: Subject:

Loop: lw.d F0, 0(R1)
add.d F4, F0, F2
sw.d F4, 0(R1)
lw.d F6, -8(R1)
add.d F8, F6, F2
sw.d F8, -8(R1)
lw.d F10, -16(R1)
add.d F12, F10, F2
sw.d F12, -16(R1)
lw.d F14, -24(R1)
add.d F16, F14, F2
sw.d F16, -24(R1)
addi R1, R1, -32 ← ۴ تا دستور را انجام دادیم
bne R1, R0, Loop

Loop Unroll: ۲۷ سیکل برای ۴ تا کار مفید

$\frac{۲۷}{۴} = ۶,۷۵$ سیکل

⇐ حل ⇐ { Loop Unroll + static sch. } ⇐ ترکیب ۲ مرحله‌ی قبل (در این با دستان بازتر است چون ۱۴ تا دستور داریم ولی در اینجا ۵ دستور بیشتر نداشتیم)

Loop: lw.d F0, 0(R1)
lw.d F6, -8(R1)
lw.d F10, -16(R1)
lw.d F14, -24(R1)
add.d F4, F0, F2
add.d F8, F6, F2
add.d F12, F10, F2
add.d F16, F14, F2
sw.d F4, 0(R1)
sw.d F8, -8(R1)
addi R1, R1, -32
sw.d F12, 16(R1)
sw.d F16, 24(R1)
bne R1, R0, Loop

هیچ وابستگی داده ای نداریم:

۱۴ سیکل برای ۴ تا کار مفید ⇐ ۳,۵ سیکل

خیلی به ۳ ایده آل نزدیک شد.

Raz

Date: Subject:

بین‌ها: ⇐ این کار حجم برنامه‌ها را زیاد می‌کند ⇐ و درنتیجه حجم حافظه را بالا می‌برد.

⇐ این کار ما برای ۷ مرحله‌ای بود ولی اگر پردازنده‌ای ساختیم که برای بهبود عملکرد پایپ‌لاین (پایپ) آن ۸ مرحله‌ای باشد ⇐ باید کل کامپایلر را از اول بنویسیم که خوب نیست.

⇐ بدی ⇐ محدودیت تعداد رجیسترها است ⇐ Register Pressure

که پس به سراغ dynamic scheduling می‌رویم ؛ چون خود پردازنده تصمیم‌گیری می‌کند و بنابراین مشکل مرحله‌ای ۷ دوم با کار را ندارد و همچنین حجم برنامه‌ها هم کم می‌شود و بنابراین مشکل سوم هم کم می‌شود که (خود کامپایلر تغییر می‌کند)

که اجرای دستورات :

پردازنده عادی

① دستور Fetch می‌شود. ⇐ این تمام دستورات بعدی را نگاه می‌دارد تا این مشکل رفع شود که ← پایپ‌لاین را متوقف می‌کند.

چند برابر هازارد ←

② اپرندهای آن خوانده شده در صورت عدم وابستگی دستور اجرا می‌شود. مشکل ما هم از همین‌جاست که پس باید کاری کنیم

که در این هم structural و هم Data هازارد داریم چون که یک جا حساب می‌کند پس پایپ‌لاین را در هر ۲ صورت متوقف می‌کند.

پردازنده با اجرای خارج از ترتیب : (out_of_order_exec)

① واحد Fetch دستورات را از حافظه خوانده و در یک صف دستورات می‌نویسد.

② اگر یک دستور به یک واحد اجرایی نیاز داشته باشد و این واحد اجرایی مشغول باشد

Structural Hazard → توقف پایپ‌لاین ← راهی جز این نداریم.

مشکل سخت افزاری

Raz

Date: Subject:

۲) اگر یک دستور اپرند لازم برای اجرا را نداشته باشد نیازی به توقف پایپ لاین نیست. در این حالت دستورات بعدی می توانند (در صورت فراهم بودن اپرندها) اجرا شوند، درحالیکه دستور قبلی منتظر حاضر شدن اپرندهای خود است.

باید واحدهای اجرایی متعدد باشد تا کار موازی را بتوان انجام داد.

۱۱، ۹، ۱۲ مسیر داده را برای اجرای دستورات خارج از ترتیب رسم می کنیم.

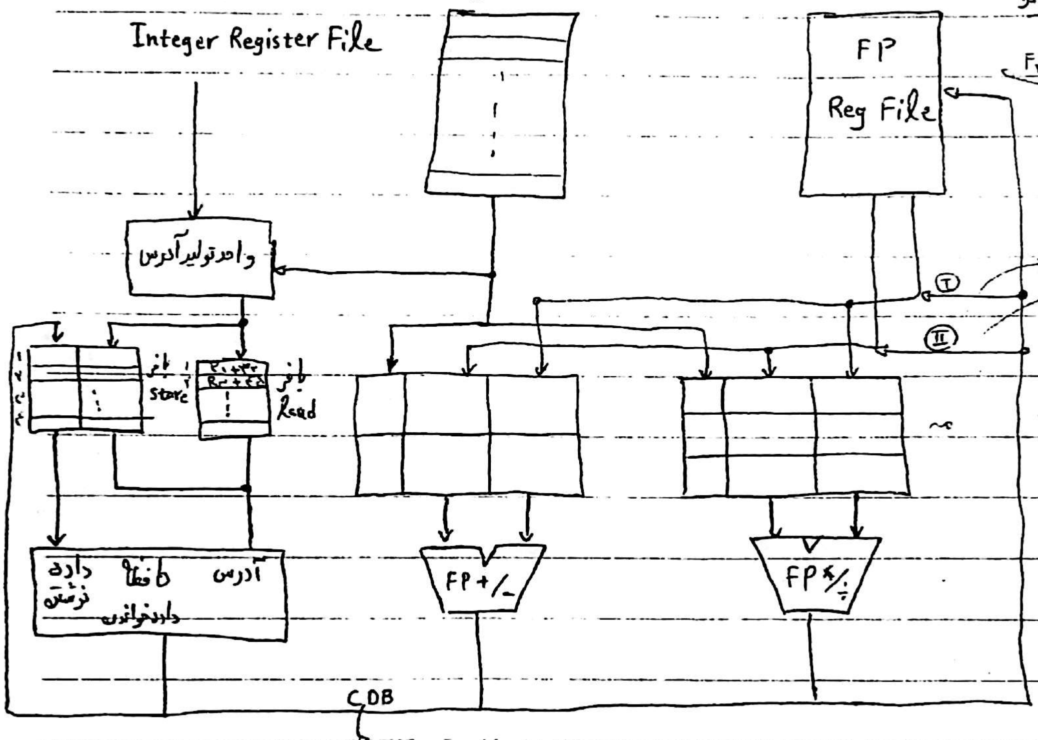
(برای اینکه بتوانیم این کار را بکنیم گفتیم که باید چند واحد اجرایی داشته باشیم.)

Load F2, 100(R1)

با صورت بلوک دیاگرام

الگوریتم توماسولو

یک واحد fetch داریم که این را خوانده و در این صف قرار داده.



می رود در F2

برای اینکه اگر بخواهیم از اپرندها دوباره استفاده کنیم

reservation station (RS)

Copy از این هم از مسیرهای I و II در RS copy می شود.

سایز RS می تواند متفاوت باشد (در ضمن می توان از ALU های بیشتری استفاده کرد ولی پیچیدگی سخت افزاری بالا می رود) ← کنترلر هم پیچیده می شود.

Raz

Date: Subject:

① اگر RS مربوط با یک واحد اجرایی پر باشد Structural Hazard رخ می دهد و پایپ لاین متوقف می شود.

② در غیر این صورت آن را با همراه اپرندهای مورد نیاز در RS مربوطه ذخیره می کند ⟸ issue

در رابطه با RS:

Name	Busy	op	V_j	V_k	Q_j	Q_k	A
نام RS entry	yes Y / NO N	نوع عملیات (جمع یا تفریق - ضرب یا ÷)					برای آدرس دستورات sw و lw است.

برای اپرند دوم (V_k, Q_k)

V_j: اگر صفر باشد: اپرند مورد نظر (در این جا ذخیره می شود و آن را داریم)
" نباشد: با دستوری که قرار است آن را تولید کند اشاره می کنیم.

Q_j: آیا اپرند داده حاضر هست یا نه

کدام دستور این رجیستر را عوض می کند:
F0
F2
F4
⋮
F30

می خواهیم این دستورات را در مسیر داده قبل چک کنیم و مراحل را ببینیم:

صف دستورات

lw.d F6, 32(R2)
lw.d F2, 45(R3)
mul.d F0, F2, F4 — وابستگی داده ای
sub.d F8, F6, F2
div.d F10, F0, F6
add.d F6, F8, F2

Raz

فرض می‌کنیم هر ۲ تا دستور ابتدا در RSها، issue می‌شوند و بعد با ترتیب شروع به اجرا شدن می‌کنند (برای سادگی؛ در عمل این گونه نیست و بلافاصله اجرا می‌شود)

Date: Subject:

فرض با فرلود ۲ تا entry داریم / برای ورودی RS+ ، ۲ تا entry / RS* ، ۳ تا entry

Name	Busy	OP	V_j	V_k	Q_j	Q_k	A
Load1	~~Y~~ N						~~R1+32~~
Load2	~~Y~~ N						~~R2+45~~
Add1	~~Y~~ N	~~-~~	M[R..] (Load1)	M[R..] ~~Load2~~	~~Ø~~	~~Ø~~ *	
Add2	~~Y~~ N	~~+~~	Add1	Load2	~~Ø~~ 1	~~Ø~~ 1	
Mul1	~~Y~~ N	~~*~~	(I) M[R..] Load2	~~F4~~	اپرند حاضر نیست ~~Ø~~ *	اپرند حاضر است +	
Mul2	Y	÷	Mul1	(Load1)	Ø	~~Ø~~ 1	
Mul3	..						

اگر بعد از این یک load بیاید پایپ‌لاین متوقف می‌شود

تغییرات پس از اجرای دستورات با ترتیب است. (خود کار مثلی، همان دستورات اولیه است).

F0	Mul1	
F2	Load2	
F4		
F6	~~Load1~~	Add2
F8	~~Add1~~	
F10	Mul2	
F12		
F14		
⋮		
F30		

مثلاً داشتن یک copy در V_k و V_j از اتفاق افتادن Hazardهایی که از Name‌ها بود جلوگیری می‌کند.

چون ما ذخیره‌ای از مثلاً F4 را داریم.

(I) یعنی این است که اپرند هنوز حاضر نیست و باید صبر کنیم تا دستور Load تمام شود و سپس نتیجه‌ی آن را یعنی طبق Load2 در V_j ذخیره کند.

چون ترتیب دستورات را داریم پس می‌توانیم قبلی را نادیده بگیریم و همچنین چون یک copy از F4 در RSها هم علاوه بر FP Reg File داریم و نیازی به update دوباره نیست.

فرض:

- +/- : 1 c.c.
- * : 10 c.c.
- ÷ : 20 c.c.

که چون sub، c.c. آن کمتر است پس زودتر اجرا می‌شود و بعد Add اجرا می‌شود و بعد Mulها و بعد div پس ترتیب اجرای دستورات تغییر می‌کند.

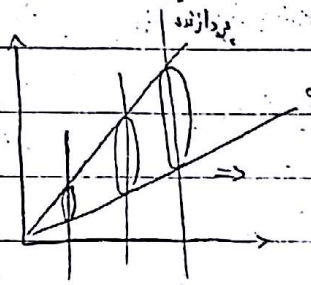
* به این روش score wording هم می‌گویند.

Raz

پایان طراحی پردازنده به صورت پایپ‌لاین

Date: Subject:

نرخ رشد سرعت حافظه ها بامراتب کمتر از پردازنده هاست ← بنابراین همین مورد محدودکننده ی سرعت پردازنده هاست.

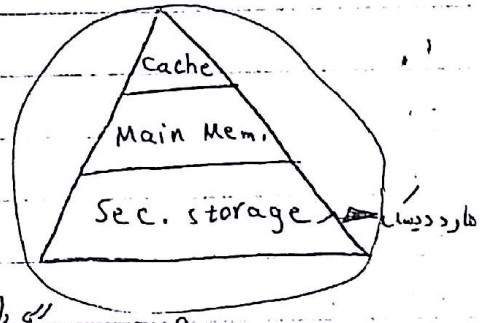


⇒ Memory wall

⇐ این فاصله بامراتب در طول زمان زیاد می شود.

Memory Hierarchy ← سلسله مراتب حافظه

- (حافظه ها را ما می توانیم سریع بسازیم ولی خیلی گران می شود)



" با ترکیبی از سخت افزار / نرم افزار می خواهیم دقت از بیرون نگاه کنیم با گنجایش درحد گنجایش هارد دیسک و سرعتی درحد سرعت cache برسیم.

← از ویژگی های برنامه ما معمولاً Locality of Ref.

① Temporal Locality ← اگر بایت خانه ای حافظه دسترسی پیدا کنیم با احتمال زیاد در آینده ی محل بودن زمان نزدیک مجدداً با آن خانه دسترسی پیدا خواهیم کرد.

② Spatial Locality ← " " " " " " " محل بودن مکان

با این ۲ تا ویژگی می توان بازدهی را بالا برد.

" " خانه های مجاور آن نیز دسترسی پیدا خواهیم کرد

Raz

Date: Subject:

پردازنده اول باسراغ Cache می‌رود اگر اطلاعات در آن نبود باسراغ Main Mem می‌رود و اگر اطلاعات در آن باشد چون احتمالاً درباره با آن اطلاعات نیاز داریم پس یک copy در Cache نیز می‌دارد و دوباره اگر درخواست main mem هم اطلاعات وجود نداشت باسراغ هارددیسک می‌رویم که در این صورت زمان خیلی بالا می‌رود.
⇐ معمولاً اطلاعات در Cache با دست می‌آید ⇐ ۹۹٪ موارد.

۹۲,۹,۱۶

Hit : دسترسی موفقیت آمیز با Cache

Miss : دسترسی ناموفق " "

$$\text{Hit Rate} = \frac{\text{تعداد Hit ها}}{\text{تعداد کل دسترسی‌ها با حافظه}}$$

$$\text{Miss Rate} = \frac{\text{تعداد Miss ها}}{\text{تعداد کل دسترسی ها با حافظه}}$$

$$\text{Hit Rate} + \text{Miss Rate} = 1$$

می‌خواهیم ببینیم چجوری می‌تونیم داده‌ی مورد نیاز در Cache هست یا نه و چون نوع داده را نمی‌دانیم!

جلسه‌ی قبل گفتیم ابتدا باسراغ Cache می‌رویم (زمان این بزند) و می‌بینیم data در آن هست یا نه سپس باسراغ Main Memory می‌رویم و ...

Cache : h_1: Hit Rate, t_1: Access Time

Main Memory : h_2: Hit Rate, t_2: Access Time

$t_1 \ll t_2 \ll t_3$

t_3: " " : HDD

Raz

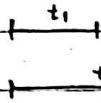
Cache, main Mem

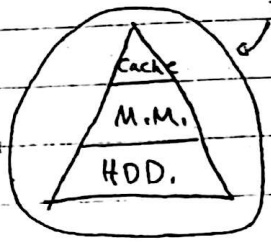
دراین جا هر ۲ باهم کار می‌کنند

اگر دیتا در Cache بود دیگر باسراغ M.M نمی‌رویم.

Date: Subject:

دسترسی موازی — دسترسی سریال — Cache اول و بعد Main Memory





زمان دسترسی پایین — نامناسب ⇐ زمان دسترسی بالا

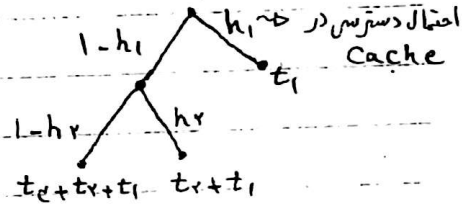
مناسب ⇐ توان مصرفی پایین — نامناسب ⇐ توان مصرفی بالا

اگر نوع دسترسی با حافظه گفته نشد ما بصورت پیش‌فرض سریال درنظر می‌گیریم.

سریال:

$$t_{eff} = h_1 t_1 + (1-h_1)\left[h_2(t_2+t_1) + (1-h_2)(t_1+t_2+t_c)\right]$$

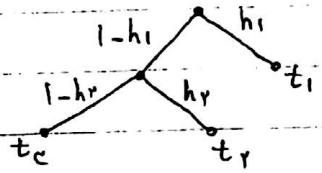
effective



اگر $h_1 \simeq 1$ ⇐

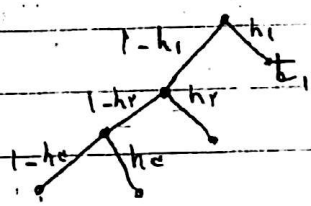
موازی:

$$t_{eff} = h_1 t_1 + (1-h_1)\left[h_2 t_2 + (1-h_2)t_c\right]$$



⇐ سطوح cache می‌تواند بیش از یک باشد. با شاخه‌های درخت قبل از M.M افزوده می‌شود

برای یک cache داریم:



پردازنده

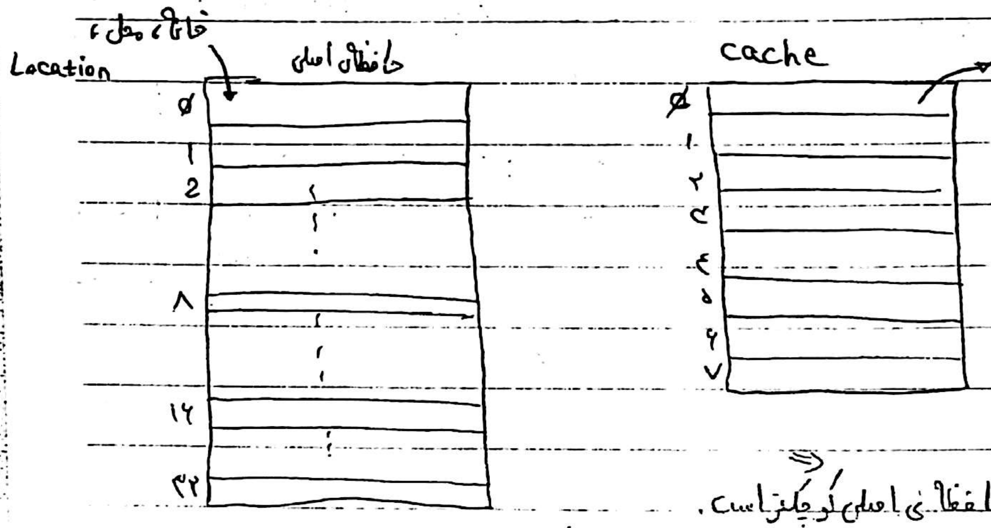
$$adr \rightarrow \begin{cases} hit \\ Miss \end{cases} \Rightarrow f(adr) = \begin{cases} 1 \rightarrow hit \\ 0 \rightarrow Miss \end{cases}$$

Raz

Date: Subject:

مکانیزم نگاشت در cache :

① نگاشت مستقیم Direct Mapping ← یک مثال ساده



⇐ cache همیشه از حافظه ی اصلی کوچکتر است.

n > k ⇐ اصل لانه ی کبوتری
(n: کبوتر، k: لانه)

{ تعداد بلوک های cache (mod) آدرس حافظه ی اصلی = شماره ی بلوک موردنظر در cache }
(mod: با تقسیم کردن)

	tag	index			
000: 0	=	00 / 0 (000)	0	mod	8
001: 1	=	00 / 1 (001)	1	mod	8
111: 7	=	00 / 111	7	mod	8
000: 0	=	01 / 000	8	mod	8
000: 0	=	10 / 000	16	mod	8
000: 0	=	11 / 000	24	mod	8

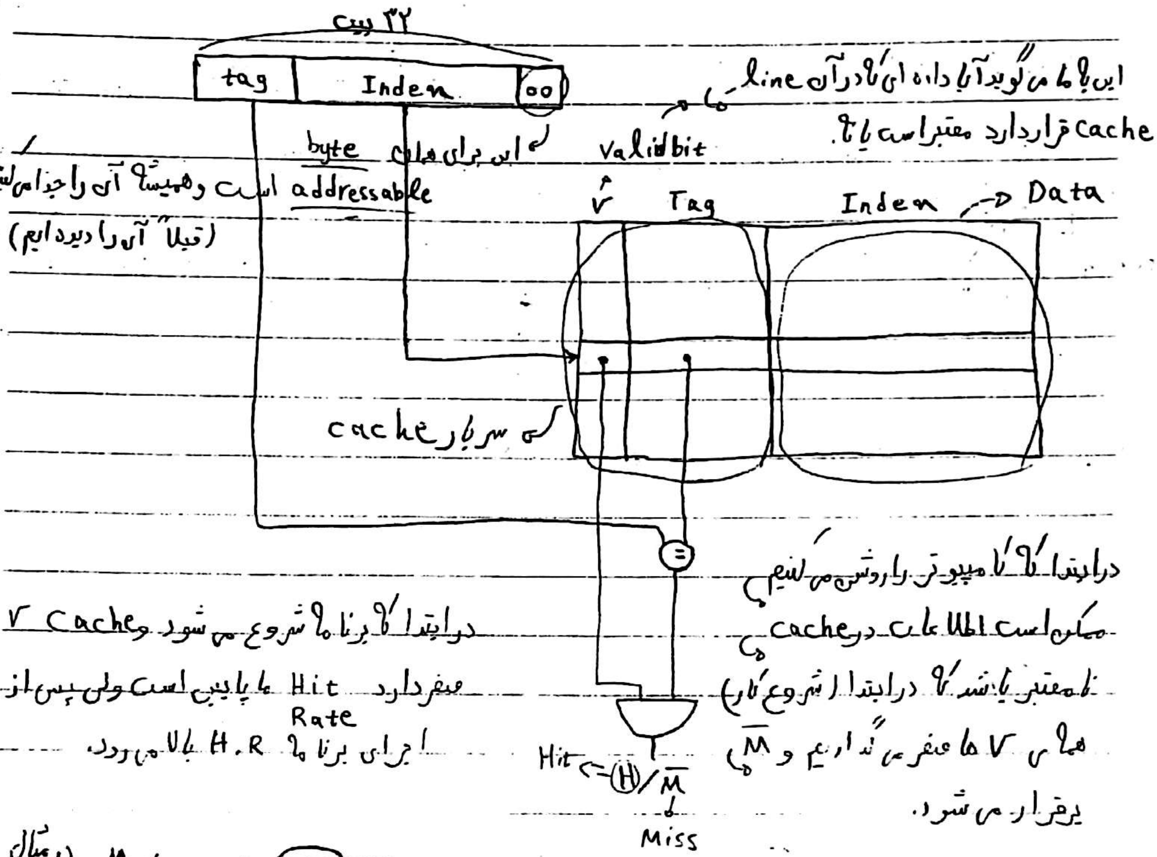
با تعداد بیت های مورد نیاز برای آدرس دهی به cache از سمت راست آدرس حافظه ی اصلی جدا کرده و آن را index می نامیم (در این جا cache، ۸ خانه داریم که با ③ بیت نیاز داریم)

می بینیم که tag آدرس هایی که در ~~خانه های یک~~ یک خانه cache قرار دارند با هم فرق دارند.

Raz

۱۰۸ وقتی می‌گویند cache ما FMB است گنجایش بخش Data فقط می‌باشد و سربار در آن محسوب نمی‌شود در MIPS

Date: Subject:



در ابتدا که برنامه شروع می‌شود cache V صفر دارد Hit Rate ها پایین است ولی پس از اجرای برنامه H.R بالا می‌رود.

در ابتدا که کامپیوتر را روشن می‌کنیم ممکن است اطلاعات در cache نامعتبر باشد که در ابتدا (شروع کار) همه V ها صفر نداریم و M برقرار می‌شود.

در مثال قبل ما	M 0	00000		
	M 5	00101		
	M 17	10001		
چون H = 0 هم V	0	00000		
یکی است	M 14	01110		
و هم tag یکی است	M 16	10000		
پس Hit داریم	H 17	10001		
	H 5	00101		

در ابتدا همه صفرند

محتویات خانه‌ای که در خانه‌های اصلی

	V	tag	
000	~~0~~ 1	~~00~~ 10	~~M[0]~~ M[16]
001	~~0~~ 1	10	M[17]
010	0		
011	0		
100	0		
101	~~0~~ 1	00	M[5]
110	~~0~~ 1	01	M[14]
111	0		

V یکی است ولی tag فرق دارد پس M[16] با tag آن overwrite می‌شود.

Hit = ۳ تعداد H ها
Rate ۸ تا دسترسی ۸ ها فقط

Raz

یعنی ۳۲ بیتی هستیم با ۲ بیت جبرا کردن و Inden و tag

Date: Subject:

مثال: در پردازنده‌ی MIPS یک cache با گنجایش ۴MB در اختیار داریم میزان

سربار cache را حساب کنید.

word برای ما مهم است

$$\frac{4MB}{4} = 1M(W) = 2^{10} \times 2^{10} W = 2^{20} W$$

خط آدرس cache

۱۰ ← ۲۰ → ۲ ← byte offset

Tag	Inden	

تعداد خانه‌های جا cach

$$2^{20} \cdot (1 + 10) = 11 \ MBit$$

(۱: v، ۱۰: tag)

برای پیاده‌سازی این cache که ما به این تعداد بیت سربار برای مدیریت آن نیاز داریم.

direct Mapping ← خوب است و ارزان هم هست ولی ما همه‌ی آدرس‌هایی با Inden یکسان را درون یک تا داریم که برای سیستم زیر خیلی بد است در حالی که می‌توانستیم آن را مثلاً در ۱۰۰ قرار دهیم که در یک بلوک خاص Map می‌کنیم

M	∅	00000
M	۸	01000
M	∅	00000
M	۸	01000
M	∅	00000
M	۸	01000
M	∅	00000

این هی تکرار می‌شود که مناسب نیست.

	V	Tag	
000	1	0001	~~M[0]~~ M[8]
001	∅		
	∅		
	∅		
	∅		

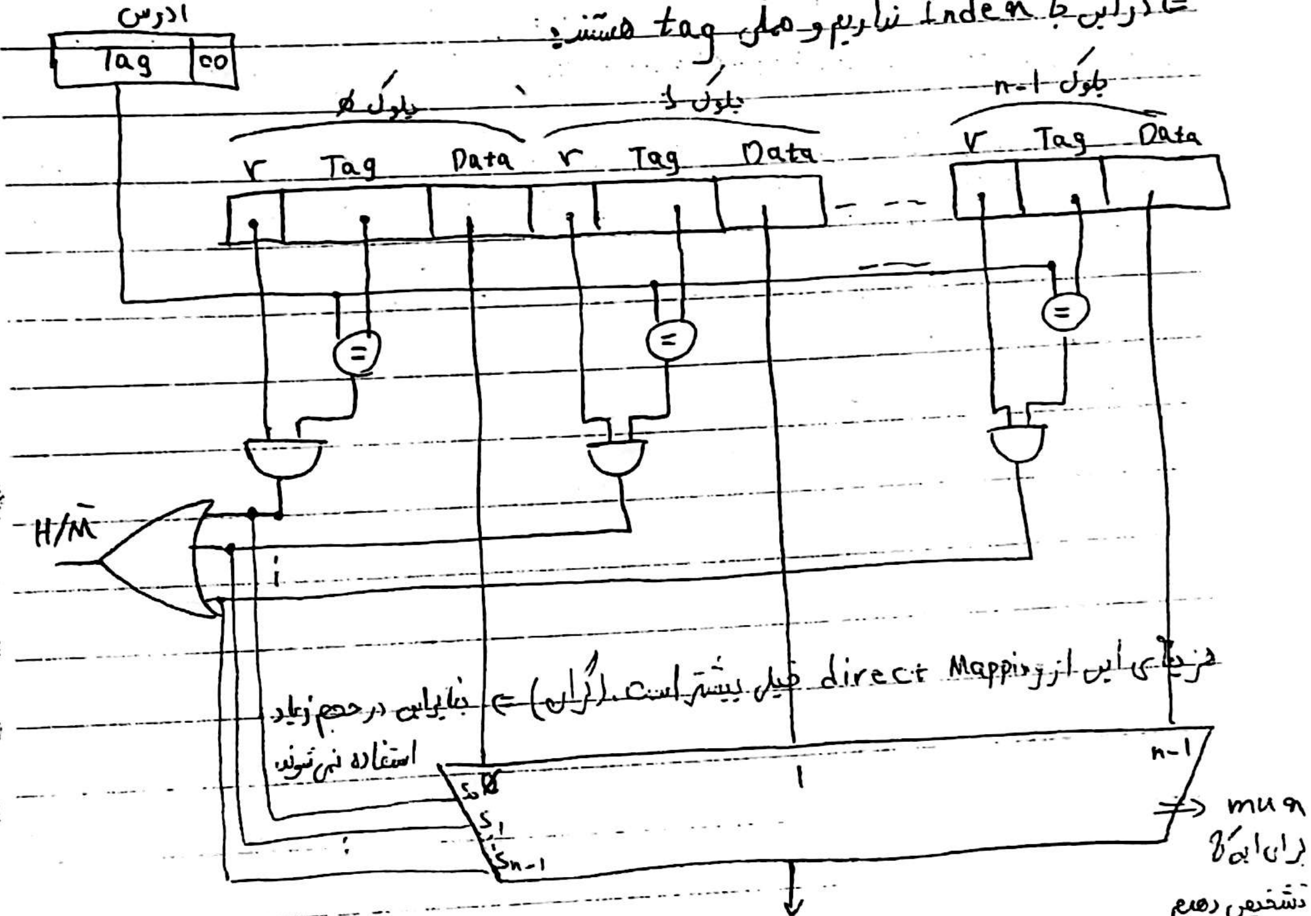
پس با سراغ بعدی می‌رویم

② تلاش شرکت‌پذیری کامل "Fully Associative" یا Content Addressable Memory (CAM)

در این جا هر آدرس در هر بلوک cache می‌تواند بنشیند ولی ما باید بصورت موازی بتوانیم در cache جستجو کنیم چون در هر کدام از خانه‌های آن می‌تواند باشد و اگر بخواهیم یکی یکی (سریال) جستجو را انجام دهیم وحشتناک می‌شود.

Date: Subject:

⇐ در این جا Index نداریم و همه tag هستند.



هزینه‌ی این از direct Mapping خیلی بیشتر است (گران) ⇐ بنابراین در حجم زیاد استفاده نمی‌شوند

mux برای اینکه تشخیص دهیم کدام data را بفرستیم (در قبلی این مشکل را نداشتیم) ⇐ S0 تا Sn-1 در واقع وقتی یک می‌شوند که hit باشند.

در هر لحظه فقط یکی از and ها یک است چون هیچگاه ۲ خانه‌ی حافظه‌ی یکسان در cache وجود ندارد چون ما خودمان داده را در cache قرار می‌دهیم و اول search می‌کنیم که آن داده داخل cache نباشد و سپس data را قرار می‌دهیم.

مثال) روش fully As. ⇐ رفتن M رخ داد آن را در اولین خانه‌ی خالی cache قرار می‌دهیم. چون به صورت موازی همه را search کردیم رد هیچکدام نبود.

از بین این‌ها M8 اخیراً کمتر استفاده شده است.

M 0 00000
M 8 01000
M 5 00101
H 0 00000
M 6 00110
M 7 00111
Raz 0 00000
H M

	V	Tag	Data
00	~~0~~ 1	00111 ~~00000~~	~~M[0]~~ M[7]
01	~~0~~ 1	00111 ~~01000~~	~~M[8]~~ M[7]
10	~~0~~ 1	00101	M[5]
11	~~0~~ 1	00110	M[6]

Date: Subject:

وقتی با ۷ می‌رسیم همه‌ی خانه‌های cache پر هستند کدام را بیرون اندازیم؟

Replacement policy: ← سیاست جایگزینی

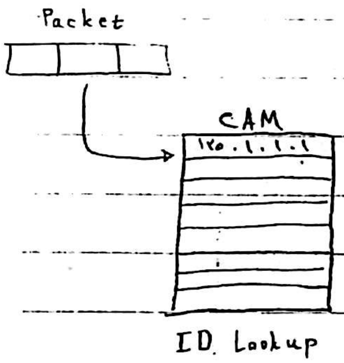
۱/۷ ** FIFO ← این می‌گوید آن آدرسی که زودتر (اول) آمده بیرون برود (از همه زودتر)

۲/۷ ** LRU ← Least Recently Used ← این می‌گوید آن که اخیراً کمتر استفاده شده است ← آن را بیرون بینداز که اخیراً کمتر به آن دسترسی داشته‌ایم.

⇐

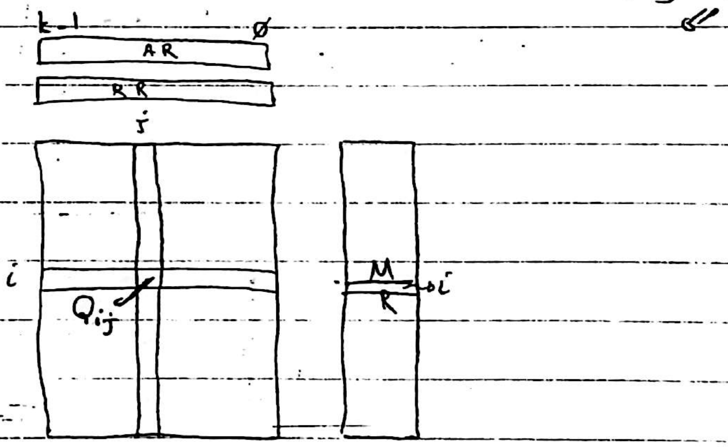
* معمولاً LRU جواب بهتری می‌دهد.

۹۲/۹/۱۸



CAM ← در سوئیچ‌های شبکه هم کاربرد دارد. Content Addressable Memory ← یک حافظه‌ی معمولی هم هست یعنی علاوه بر نوشتن و خواندن در داده‌های آن امکان جستجو هم داریم.

ساختار CAM



AR: Argument Reg.

KR: Key "

MR: Match "

Date: Subject:

مثال)

AR	00101
KR	00111

	MR
10101	1
11101	1
00101	1
11001	0
11111	0
01001	0

⇐ داده ای است که می خواهیم جستجو کنیم ⇐

در این جا می خواهیم داده ای را جستجو کنیم تا ۳ بیت سمت راست آن 101 است ⇐ هر کدام از KR بیت هایش صفر باشد آن بیت در AR را skip می کند و جستجو نمی کند پس طبق این قاعده ۳ تای اول ها Match هستند و در MR ، یک شده اند.

match بودن ستون را نشان می دهد — Match بودن سطر را نشان می دهد.

طبق صفحات قبل ⟵ $M_{ij} = \overline{KR_j} + KR_j\,(\overline{AR_j}\,\overline{Q_{ij}} + AR_j\,Q_{ij})$

(زیر KR_j: 1 — زیر جمله ی اول: هر ۲ صفر — یا — زیر جمله ی دوم: هر ۲ یک)

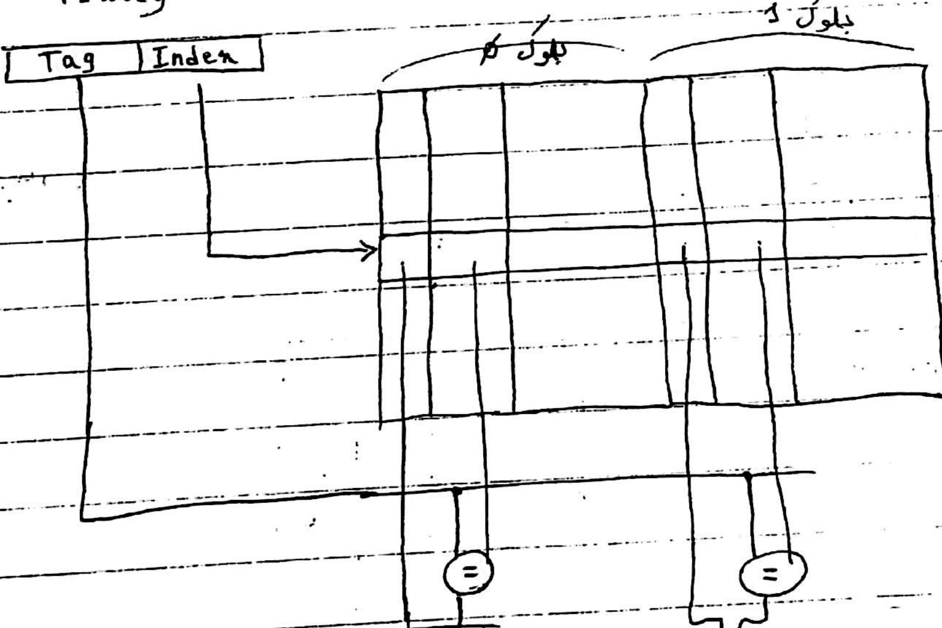
که این همان skip کردن است که گفتیم، اگر یک مقدار KR صفر بود ستونش را نمی خواهد برای Match بودن با AR چک کنی

$$MR_i = \underset{j=0}{\overset{k-1}{\text{AND}}}\; M_{ij}$$

یعنی همه ی داده های یک سطر باید Match باشند.

③ مجموعه ای انجمنی و Set-Associative (تقریباً ترکیب ۲ تای قبل است)

تعداد بلوک های داخل یک مجموعه ⇓ 2-way



ابتدا یک مجموعه را انتخاب می کنیم سپس در آن مجموعه به صورت موازی جستجو می کنیم.

بقیه ی مدارش یعنی OR و mux ⇐ مثل مدار مدل ② است و همچنین سیاست جایگزینی، هم با یکی از ۲ روش گفته شده داریم.

Date: Subject:

مثال) یک حلقه ۳ بار تکرار می‌شود. در هر بار تکرار آدرس‌های ۰ تا ۴۴ تولید می‌شود. یک cache با گنجایش ۴۴ کلمه داریم.

الف) Direct Mapping

ب) Fully Ass. (FIFO)
سیاست جایگزینی در صورت نیاز

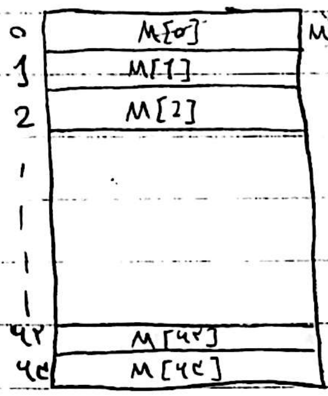
ج) 2-way set-Ass. (FIFO)

برای یافتن خانه‌ی حافظه باقیمانده بر ۴۴ را می‌یابیم.

Miss

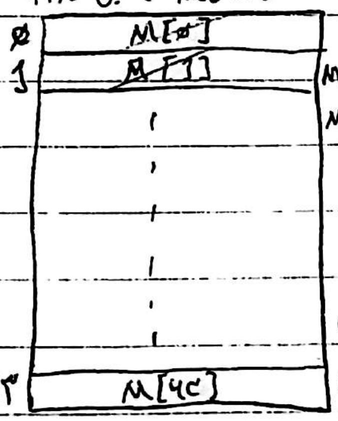
0 M M M | 0 M M M | 0 M M M
1 M M M | 1 H M H | 1 H M H
2 M M M | 2 H M H | 2 H M H
۴۰ M | ۴۰ H | ۴۰
۴۱ M | ۴۱ H | ۴۱ → H
۴۲ M | ۴۲ → M | ۴۲ → M
۴۳ M | ۴۳ → H | ۴۳ → H
۴۳ M M M | ۴۳ H M H | ۴۳ H M H
۴۴ M M | ۴۴ M M M | ۴۴ M M M

الف)
M[44]
M[0]
M[44]
M[0]
M[44]



$$\text{Hit Rate} = \frac{2\times 43}{3\times 45}$$

ب)
fifo طبق ⇐ M[44]
در جای اول



M[0] ⇒ Miss چون FIFO هستیم پس همه Miss می‌شوند (جای اولین وارد شده قرار می‌گیرند.) زودتر وارد شده

Hit Rate = 0

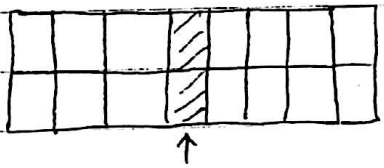
Date: Subject:

	M[۰]	M[۴۴]
۰	M[۰]	M[۴۴]
۱	M[۱]	M[۴۵]
۳۱	M[۳۱]	M[۷۵]

در این روش Mod با ۳۲ را حساب می کنیم.

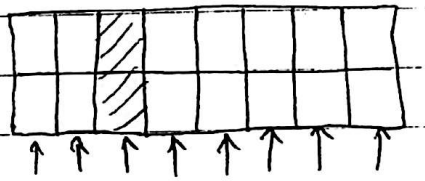
$$\text{Hit Rate} = \frac{۲ \times ۴۲}{۴ \times ۴۵}$$

↳ به صورت دایگرامی:



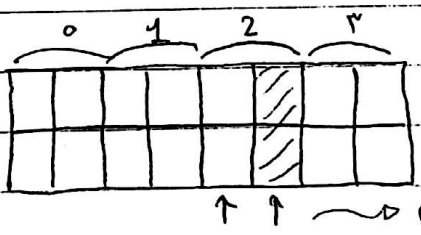
direct Mapping

(با سراغ یک بلوک خاص می رویم)



Fully Ass.

(جستجو به صورت موازی)



set _ Ass.

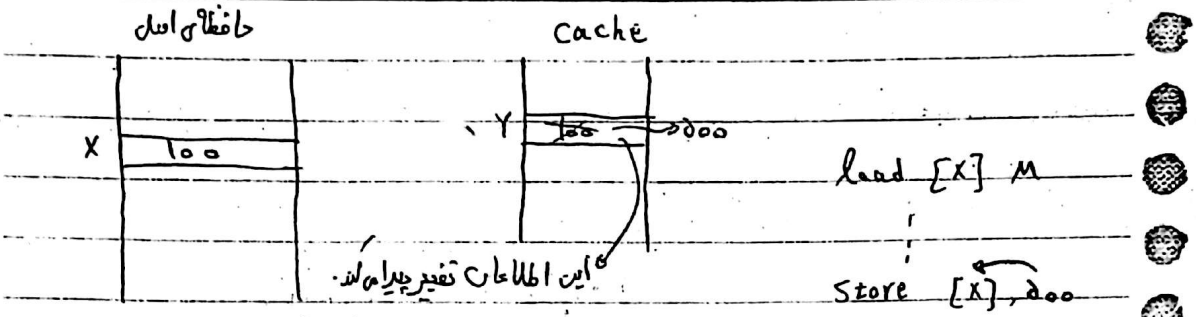
در یک مجموعه از بلوک ها به صورت موازی جستجو می کند.

نقش حافظه نهان در نوشتن؟

(Cache)

Raz

Date: Subject:



ناسازگاری cache (cache Incansistency): اگر محتویات یک خانه‌ی حافظه‌ی اصلی با کپی متناظر آن در cache یکسان نباشد می‌گوییم ناسازگاری cache رخ داده است.

برای رفع مشکل بالا: writing policy

① write Through: هر نوشتن در cache با یا هنگام کردن کپی آن در حافظه‌ی اصلی همراه شود ⇐ این کار استفاده از cache را بی معنی می‌کند و سرعت را بسیار پایین می‌آورد چون دوبار باید با طور مرتب دسترسی با حافظه داشته باشیم.

راه دوم برای رفع این مشکل این است با سرعت cache قرار می‌دهند [البته اگر باز پر شود ۲ بار اختلاف سرعت ایجاد می‌شود ولی چون دستورات Store کم است مشکلی ایجاد نمی‌شود]

آدرس	X	
داده	۵۰۰	

② write Back: فقط با هنگام خارج شدن محتویات یک بلوک cache، کپی آن را در حافظه‌ی اصلی یا هنگام می‌کنیم ⇐ این ناسازگاری را می‌پذیرد ولی در موقع نیاز آن را رفع می‌کند یعنی تا زمانی که از cache می‌خوانیم مشکلی پیش نمی‌آید چون داده update شده را داریم ولی وقتی می‌خواهیم محتویات cache را خارج کنیم ممکن است این داده را از دست بدهیم پس آن را با حافظه‌ی اصلی منتقل می‌کنیم.

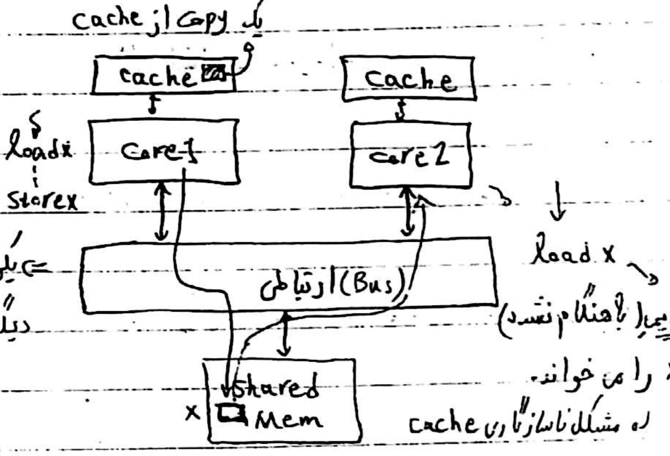
Raz

Date: Subject:

Multi-core — این ناسازگاری در Multicore رخ می‌دهد

① Message Passing مثل شبکه‌های معمولی هم‌

② shared Memory ← در این جا مشکل می‌خوریم!

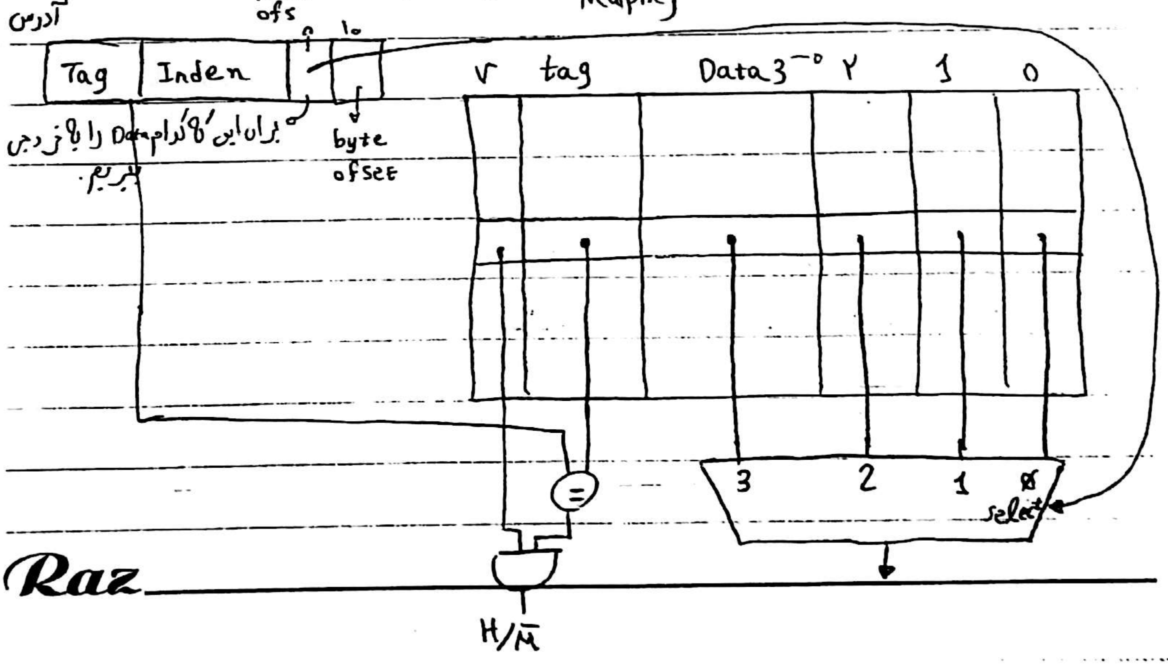


← یکی روی حافظهٔ مشترک می‌نویسد، هسته‌ی دیگه از رادون می‌خونه (ردی)

← یکی قدیمی را (هماهنگ نشده) x را می‌خوانند که مشکل ناسازگاری cache

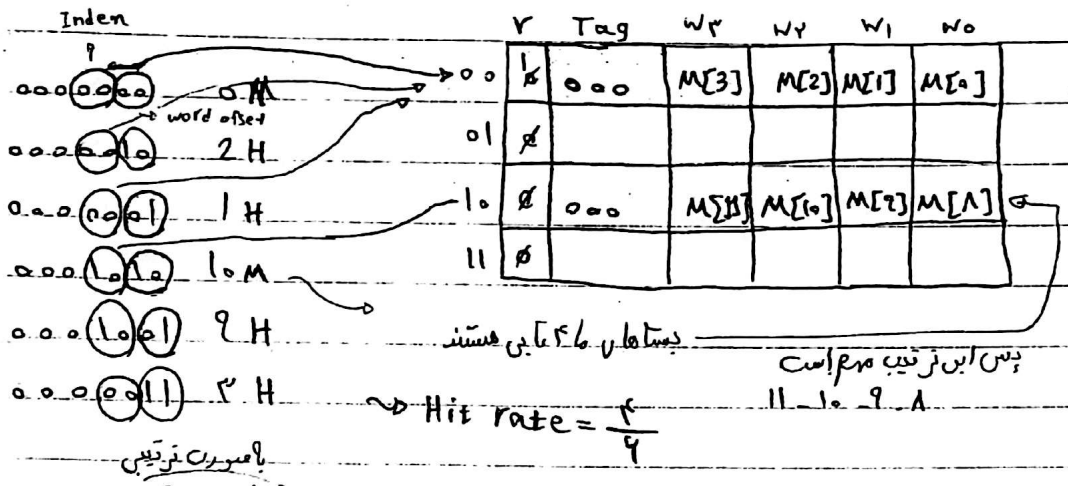
اصل locality زمانی

پس از این موارد با سراغ اصل locality مکانی می‌رویم: (بنابراین علاوه بر بلوک اصلی خانه‌های مجاورش را هم بر می‌داریم و می‌آوریم) ← تفاوت این روش با روش Set Ass. و در set Ass. هر بلوک جداگانه چک بررسی می‌شود (مثلاً در یک مجموعه هر بلوک یک V جدا دارد ولی این جا ... مثلاً هر ۴ بلوک یک V دارند) ← در این جا هم با هر یک از سه روش direct Mapping، fully و set Ass. می‌توانیم عمل کنیم، ما در این جا برای direct Mapping توضیح می‌دهیم.



Date: Subject:

مثال) ظاهای cache:



بلوک‌ها ۴۶ بایتی هستند

پس این ترتیب مهم است

۸ ـ ۹ ـ ۱۰ ـ ۱۱

بامیرت ترتیب

→ Hit rate = ۴/۶

در ۲ ط locality مثالی خیلی کاربرد دارد ← برای دسترسی با داده‌های یک آرایه و بعدی هم جایی است ط instruction های داریم ط دستور پرش نخورده ایم چون instruction ها ترتیبی اجرا می‌شن خیلی کاربرد دارد.

تا

به همین دلیل instruction و Data Cache Cache معمولاً داریم.

Raz

Date: Subject:

۹۲,۹,۲۵

مثال: یک پردازنده با $f = 5GHz$ با $CPI = 1.0$ و $T = 0.2^{ns}$

همه دسترسی‌های به حافظه موفقیت‌آمیز باشد. (حالت ایده‌آل)

حافظه اصلی ← زمان دسترسی $t = 100ns$

Level 1

L_1 cache ← Miss Rate = ۲٪

L2 cache ← $t = 5ns$, Miss Rate = ۰,۵٪

می‌خواهیم ببینیم با cache ، تا این ما چقدر افزایش می‌یابد.

$CPI = CPI_{Ideal} + Memory\_stall\ clk\ cycles\ Per\ Inst.$

جریمه دسترسی به M.M. (Main Memory) $= \frac{100^{ns}}{0.2^{ns}} = 500^{C.C}$

$CPI = 1 + 500 * 2\% = 11$ یعنی هر دستور به طور متوسط ۱۱ سیکل کلاک طول می‌کشد.

به اطلاعات در L2 در ۲٪ موارد نیست پس باید سراغ M.M می‌رویم.

جریمه دسترسی به L2 در حالت وجود در L2 $= \frac{5ns}{0.2ns} = 25^{C.C}$

$CPI = 1.0 + 2\% * 25 + 0.5\% * 500 = 4.0$

جریمه‌ی دسترسی به L2 ، جریمه‌ی دسترسی به M.M.

در L2 نیست، باید سراغ M.M. می‌رویم. در L2 cache نیست، باید سراغ L2 می‌رویم.

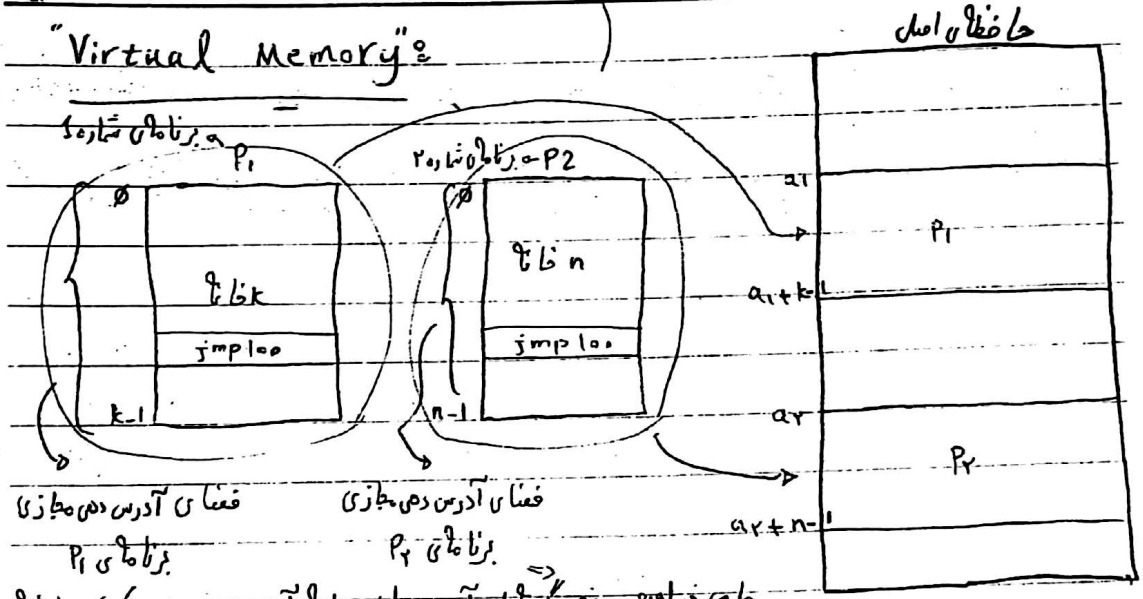
$\Rightarrow Speedup = \frac{11.0}{4.0} \sim 2.8$

Raz

یا loader در سیستم عامل، برنامه را در جایی از حافظه قرار می دهد.

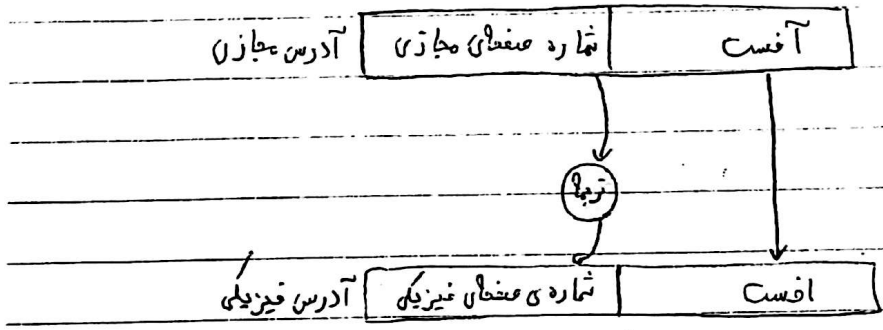
Date: Subject:

"Virtual Memory"



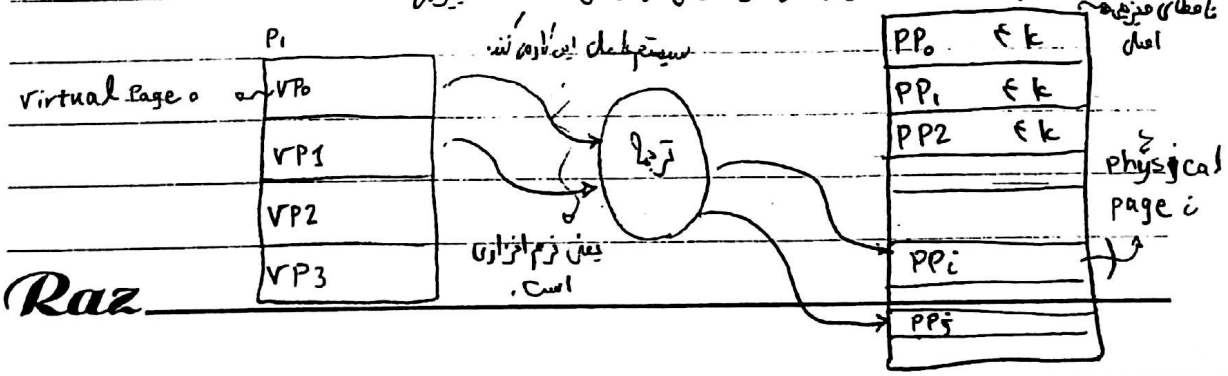
ما می خواهیم ببینیم چگونه این آدرس مجازی را با آدرس واقعی (فیزیکی) در حافظه اصلی تبدیل کنیم.

jmp 100 در برنامه ی P_1 می خواهد با ۱۰۰ برنامه ی خودش (P_1) برود و در P_2 هم همین طور پس باید مکانیزمی باشد که در حافظه ی اصلی شروع برنامه را نگاه دارد و بتواند با ۱۰۰ برنامه ی P_1 برود.



صفحه ی Page ← قطعات منطقی از حافظه ی مجازی و اصلی با آن تقسیم شده اند.

در این ترجمه، آفست عوض نمی شود ولی شماره ی صفحه تغییر می کند.

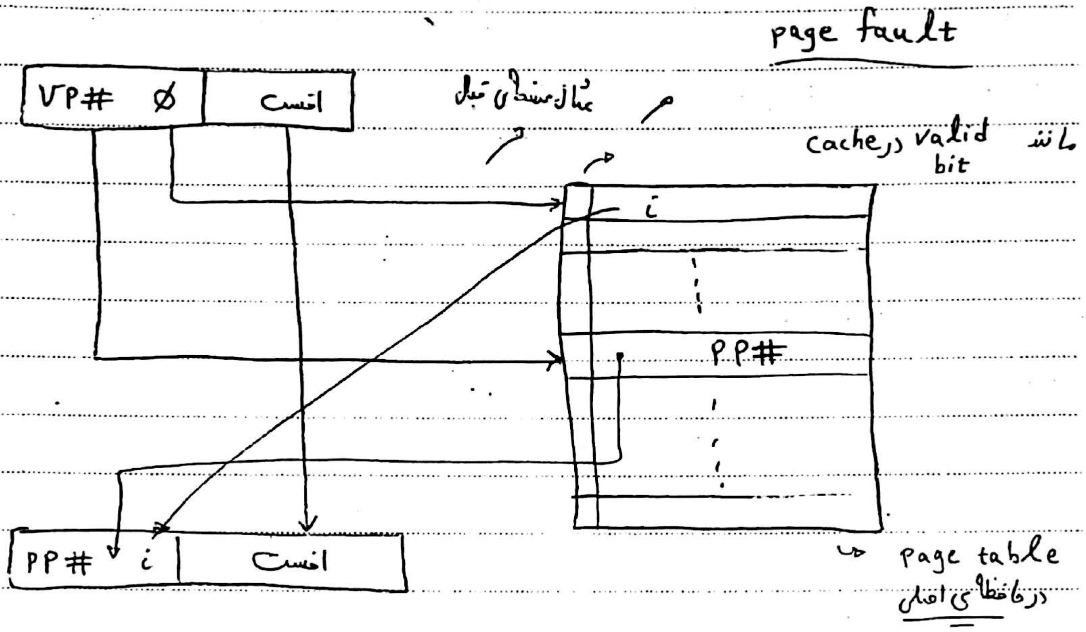


Raz

Subject
Date

ارتباط بین M.M و این برنامه‌ها نرم افزاری است و با سیستم عامل است.
که در حالی که ارتباط بین M.M و cache فیزیکی و سخت افزاری بود.

یک table ای برای این تبدیل (ترجمه) وجود دارد:

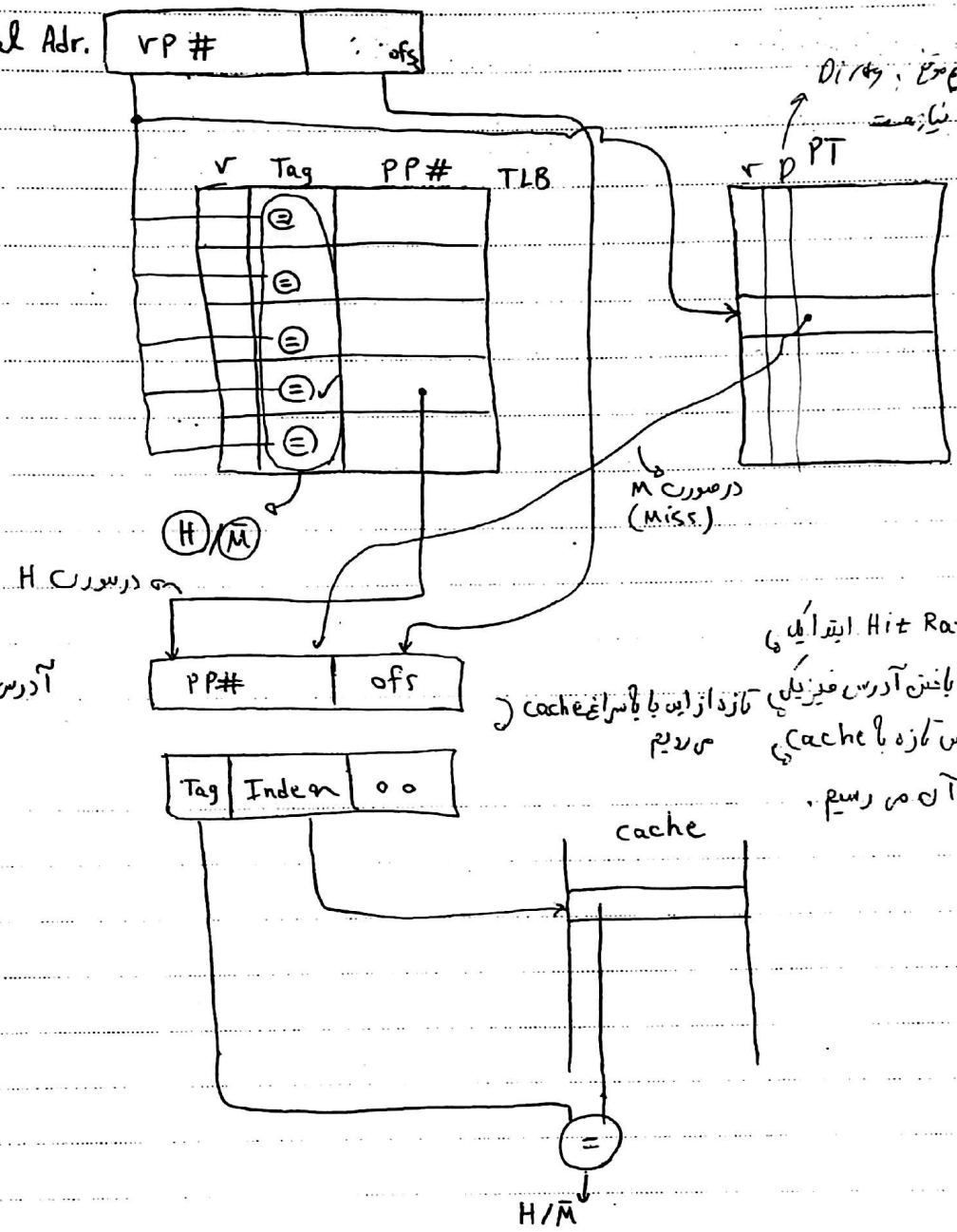


هنگامی که با یک virtual page دسترسی پیدا کنیم که هنوز در Page table ، Map نشده باشد Page fault رخ می‌دهد و سیستم عامل این Mapping را اول انجام می‌دهد و آن را مدیریت می‌کند و intry مربوط به آن را set می‌کند.

مشکل این جا این است که ما cache را برای این قرار داده بودیم که سراغ M.M نرویم ولی این جا برای این که آدرس مورد نظر را در cache بیابیم مجبوریم یک بار به سراغ M.M برویم و آدرس را بسازیم و سپس به سراغ cache برویم و اگر در آن data نبود دوباره باید به سراغ M.M برویم که این فاجعه است و خیلی بد می‌شود پس یک cache دیگر داریم به نام TLB تا این مشکل را حل کنیم.

معمولاً سایز کوچکی دارد و با مبنی ← Translation Look Aside Buffer (TLB)
دلیل معمولاً fully Asso. است، Page Table
که یک خاص cache می‌ فقط برای نگهداری بخشی از PT که زیاد مورد استفاده قرار می‌گیرد. (پس ابتدا آدرس را در این جا جستجو می‌کنیم و اگر در این نبود به سراغ حافظه‌ی اصلی و PT می‌رویم)

Subject
Date



آیا تغییر کرده‌ایم که سیستم عامل Direct خارج کرده از حافظه نیز یکی نیازمند در حافظه نیز یکی نیز هم باشد

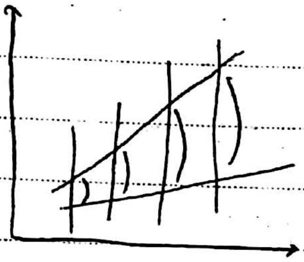
پس برای Hit Rate ابتدایی مرحله برای یافتن آدرس فیزیکی تا زدا ز این با یا سراغ cache داریم و سپس تازه با cache می‌رویم ودرون آن می‌رسیم.

حال با سراغ پردازنده‌ی Multicore می‌رویم:

Instruction Level parallelism (ILP)

پایپ لاین — افزایش performance برای افزایش کارایی پردازنده‌ها

Subject
Date



Power wall
Mobile } ⇐ این افزایش فرکانس با افزایش حرارت همراه بود که بسیار وحشتناک بود پس از یک جایی باید این روش را متوقف کرد.
ما با چراغ زیر رفتیم

⇐ با جای این که فرکانس را بالا ببریم ⇐ از چند پردازنده با فرکانس کمتر استفاده کنیم. ⇐ Task / Thread level Parallelism (TLP)

مثلاً:

	clk	کارایی	توان مصرفی
پردازنده‌ها	۱	۱	۱
با افزایش فرکانس 20% overclock	۱,۲	۱,۱۸	۱,۶
کاهش فرکانس 20% underclock	۰,۸	۰,۸	۰,۵۸
20% ″	۰,۸	۰,۸	۰,۵۸
		۱,۶	۱,۱۶

با افزایش ۶۰٪ توان مصرفی، ۱۸ درصد کارایی افزایش یافت.

که یعنی با افزایش ۱۶ درصد توان مصرفی کارایی ۶۰٪ افزایش یافت.
پس این روش به مراتب بهتر است.

که این با تنها کسی که ناراحت است، برنامه‌نویس است.

flyn's Taxonomy:

جدول در متن بعد

Subject
Date

کاربرد زیادی دارد ⇐ برای برنامه‌هایی که یک کار را زیاد تکرار می‌کنند و با طور مشابه عمل می‌کنند.

① SISD پردازنده‌های عادی

② SIMD پردازنده‌های برداری

Single / Multiple — Inst. — Single / Multiple — Data stream

مثلاً $a = (a_1\ a_2\ \cdots\ a_n)$ +
$b = (b_1\ b_2\ \cdots\ b_n)$

مثلاً پردازنده‌ی CSX 700 (با ۱۹۲ المان پردازش) بر این مبنا با طور خاص طراحی می‌شوند.

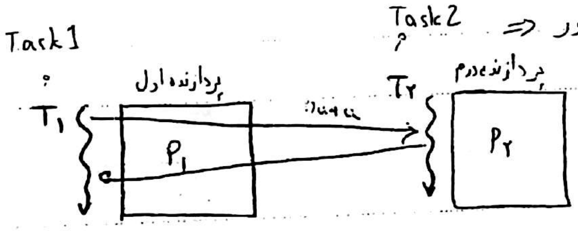
③ MISD X ⇐ کاربرد ندارد و ساخته نمی‌شود

④ MIMD ← این همان TLP هاست.

Multiprocessor: ⇐ یک سیستم کامپیوتری با بیش از ۲ پردازنده

Multi core: پردازنده‌ها روی یک تراشه قرار دارند.

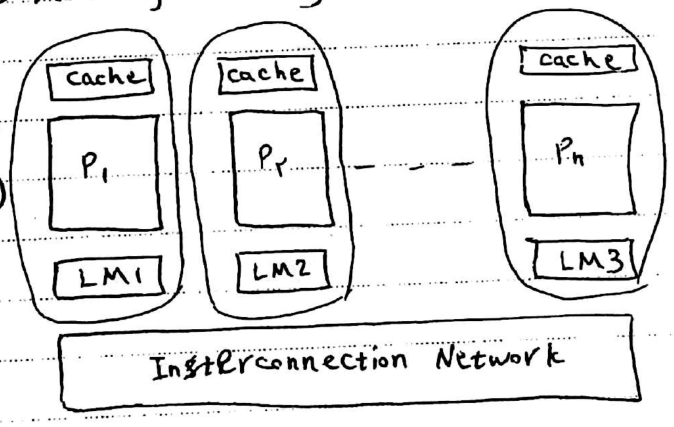
Task2 ⇐ این ۲ Task از یک برنامه هستند با طور معمول و بنابراین به Data ها هم نیاز دارند.



برای ارتباط این ۲، ۲ تا مکانیزم داریم.

① Message Passing Multi-processor   LM: Local Memory

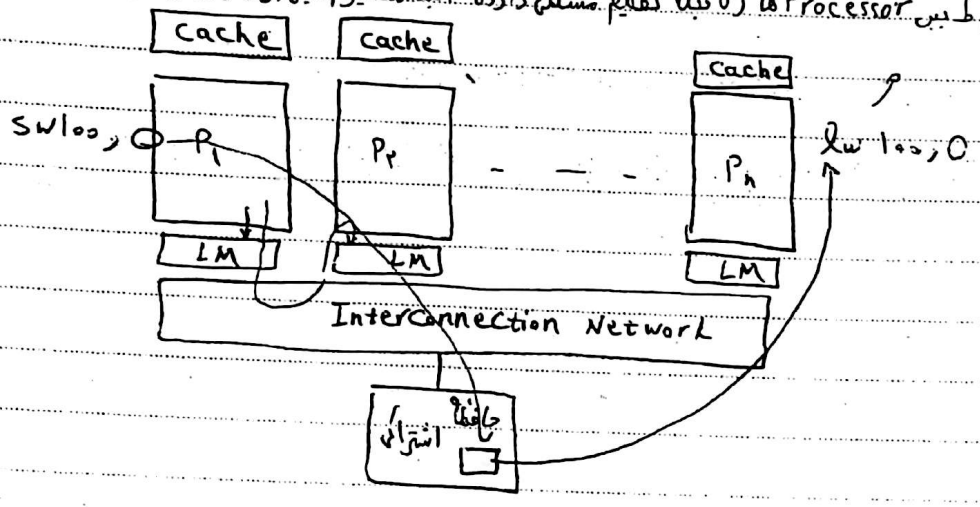
Send (P_2, O)
O = Receive (P_1)
↓
مکانیزم Send و Receive



② shared Memory Multiprocessor (SMP)

Subject
Date

ارتباط بین Processor ها (که قبلاً گفتیم مشکلی دارند) البته مکانیزم‌هایی برای رفع آن وجود دارد.



مکانیزم‌های SMP در کامپیوترهای رده بالا

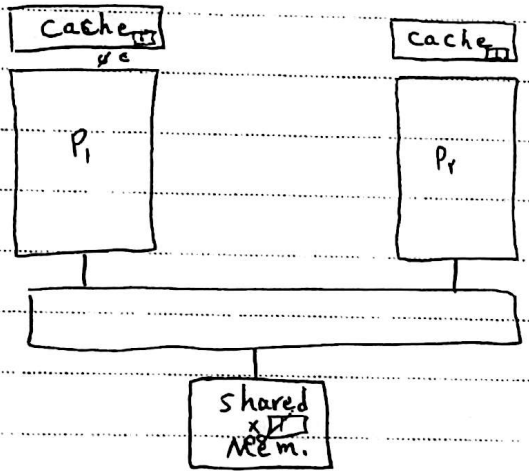
UMA (unified MeM. Access) ⇐ زمان دسترسی به حافظه برای همه پروسسورها یکی است.

Non_UMA ⇒ این، این‌گونه نیست

↳ جلسه‌ی بعد کارایی این پردازنده‌ها را بررسی می‌کنیم.

↳ مثلاً هر پروسسوری به LM خودش زودتر دسترسی دارد تا به LM دیگری یا دیگری. ⇐ چیزی مانند دسترسی‌ها یکی نیست.

۹۲/۹/۳۰



Subject
Date

زمان	event	مقدار x در cache۱	مقدار x در cache۲	مقدار x در حافظه
۱	—	—	—	۱
۲	P_1 مقدار x را می‌خواند	۱	—	۱
۳	" " " P_2	۱	۱	۱
۴	" " " P_1 صفر می‌کند	۰	۱	۰
۵	P_2 " x را بخواند			

له‌ی اشتباه است چون P_2 ، x خود را از یک صفر تغییر نداده

① Directory Based و cache های ⇐ داده‌ای که cache ها دارند را در جایی ذخیره می‌کنند و هروقت عوض شد همه‌ی cache ها را آپدیت می‌کنند.

② snooping و cache های ⇐ یک bus دارند همواره bus را نظارت می‌کنند اگر در یک P ، داده عوض شد و آپدیت شد آن هم داده‌ی خود را عوض می‌کند.

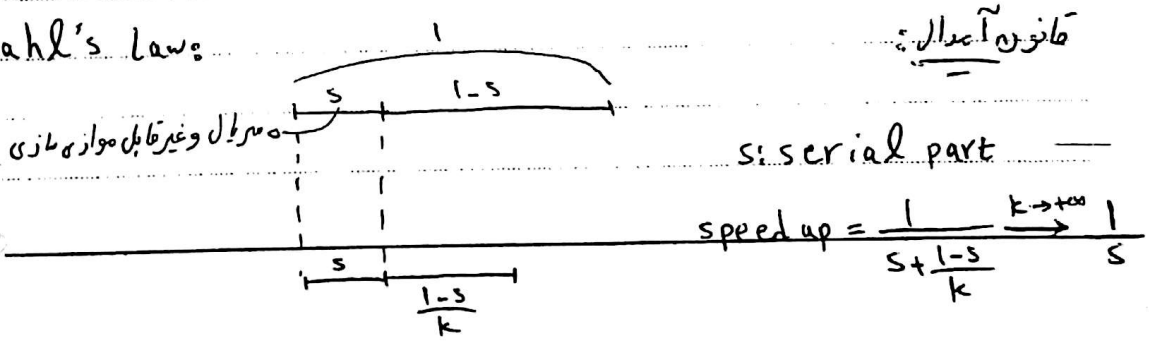
(۲-۱) write update

(۲-۲) write Invalidate این هم مانند بالا همواره نظارت می‌کند وقتی یک P ، آن کپی را آپدیت کرد ، این بیت ولید خود را صفر می‌کند پس هنگامی که P_2 خواست x را بخواند از shared mem آپدیت شده است می‌خواند.

(از پایین به بالا بیشترین کاربرد را دارند.)

Amdahl's law:

قانون آمدال:



s: serial part ← سریال و غیرقابل موازی سازی

$$speedup = \frac{1}{s + \frac{1-s}{k}} \xrightarrow{k \to +\infty} \frac{1}{s}$$

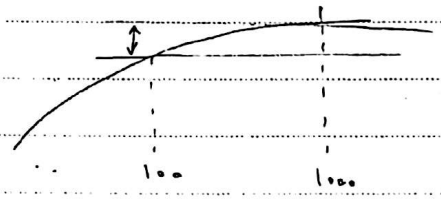
Subject
Date

$s=10\%$, $k=10 \Rightarrow speed\ up = \frac{1}{0.1 + \frac{0.9}{10}} = \frac{1}{0.19} = 5$

$k=100 \Rightarrow$ " $= \frac{1}{0.1 + \frac{0.9}{100}} = \frac{1}{0.11} \cong 9.8$

$k=1000 \Rightarrow$ " $= 9.99$

با هر پردازنده ۱۰ برابر Gain ای که در سرعت به دست آوردیم خیلی کم است و اصلاً قابل توجیه نیست.



Linear speed up

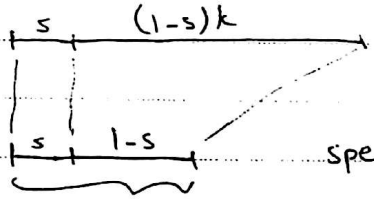
k پردازنده $\Rightarrow$ k مقدار تسریع

آمدال فرض می‌کرد طول مسأله ثابت می‌ماند در حالی که در واقعیت این‌گونه نیست پس داریم:

Gustafson's Law

سریال ← طول مسأله بیشتر می‌شود

موازی

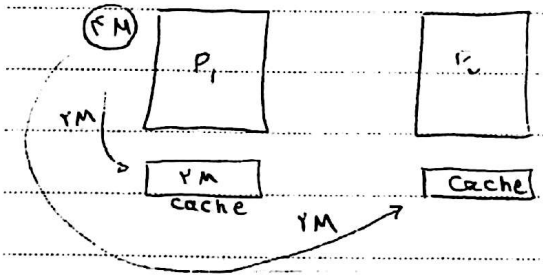


$speed\ up = s + (1-s)k$

اگر k خیلی کوچک باشد،

Linear speed up می‌تواند رخ دهد ولی در قانون آمدال این امکان پذیر نبود.

super Linear speedup ; k پردازنده $\Rightarrow$ تسریع $> k$ $\Rightarrow$ در بعضی شرایط خاص می‌تواند اتفاق افتد.



P_1 با : $T_1 = 1$ زمان اجرا + زمان انتظار برای Miss

P_2, P_1 : $T_2 = 2$ زمان اجرا + $\varnothing$

پس میزان تسریع از ۲ برابر بیشتر شد.

در صورت پردازش موازی زمان برای sweep و swap نیاز ندارد.

Subject
Date

مثال: جمع ۱۰ متغیر اسکالر و جمع ۲ ماتریس ۱۰×۱۰ ، ۱۰ و ۱۰۰ پردازنده

100×100 ، بزرگ شدن ابعاد مسأله ، t

پیش از موازی سازی : $T_1 = 10t + 100t = 110t$

speed up

موازی با ۱۰ پردازنده : $T_2 = 10t + \frac{100t}{10} = 20t$ $\quad S = \frac{110t}{20t} = 5.5$

خواسته ی تسریع ما : ۱۰ $\quad$ ۵۵٪ محقق شد

" ۱۰۰ " " : $T_3 = 10t + \frac{100t}{100} = 11t$

$S = \frac{110t}{11t} = 10$ $\quad$ ۱۰۰ $\quad$ ۱۰٪ $\quad$ این بهتر است.

خواسته ما ۱۰۰ ، ما ۱۰ ، ۱۰٪ محقق شد.

(یعنی می خواستیم سرعت ۱۰۰ برابر شود)

⇐ حال با قانون Gustafson مورد بالا را بررسی می کنیم (گستاوسن می گوید هم زمان با افزایش پردازنده ها ، پیچیدگی را هم بالا ببرو طول مسأله را هم بزرگ کن تا تسریع بیشتر شود) ابعاد

پیش از موازی سازی : $T_1 = 10t + 10000t = 10,010t$

موازی با ۱۰ پردازنده : $T_2 = 10t + \frac{10000t}{10} = 1010t$

$S = \frac{10010t}{1010t} = 9.9$ $\quad$ ۹۹٪

" ۱۰۰ " " : $T_2 = 10t + \frac{10000t}{100} = 110t$

$S = \frac{10010t}{110t} = 91$ $\quad$ ۹۱٪

⇐ پس این درصدها بسیار بهبود یافت

و آن را در بزرگ کردن ⇐ پس خیلی قابل موازی سازی نیست. ⇐ معمولاً با I/O ها مربوط ⇐ بخش سریال ابعاد مسأله را پایین آوریم. است.

☐ تسریع خوب با بزرگ کردن ابعاد مسأله به دست می آید.

Subject
Date

Load Ballancing: (توازن بار)

۲۰ برنامه؟ ۸۰ برنامه؟
P_1 P_2 ... P_{10}

می خواهیم ببینیم speed چه ⇐ یعنی توازن بار نداریم ⇐

و چه میزان تغییر می کند و هر کدام $\frac{80}{9}$ می رسد

مثال) مثال قبل با ۱۰۰ پردازنده و یک پردازنده ۲٪ کار و بقیه ۹۸٪ می دهیم (و بقیه ۱٪)

$$10000 \begin{cases} 2\% \Rightarrow 200 \\ 98\% \Rightarrow 9800 \end{cases}$$

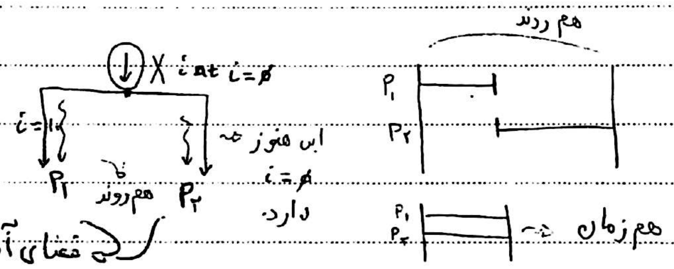
زمان پس از موازی سازی $= 10t + Max\left(200t + \frac{9800t}{99}\right) = 210t$

$$S = \frac{10010t}{210t} \simeq 48\%$$ نسبت به ٪۹۱ خیلی افت ... بدتر شده

همه برنامه هایی که در هارد دیسک نوشته ایم و دیتابیس آن، حتی که توسط پردازنده اجرا می شود. (حافظه)

برنامه ها موازی:
- Multi-process
- Multi-Thread

Creat 2 processes ⇐



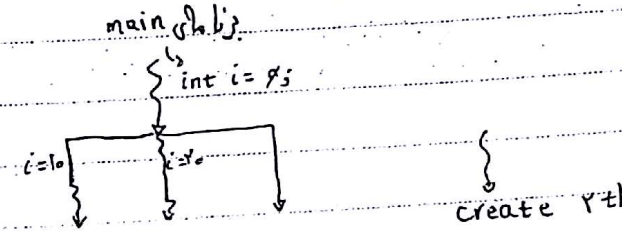
⇐ فضای آدرس دهی اختصاصی (فضای مشترک حافظه ای ندارند و هر کدام فضای آدرس دهی مخصوص به خود دارند و بنابراین برای حرف زدن باهم با message passing نیاز دارند.)

Inter-Process comm
(IPC)

Process ها خیلی سنگین هستند و OS نیاز دارد سربار زیادی اختصاص دهد.

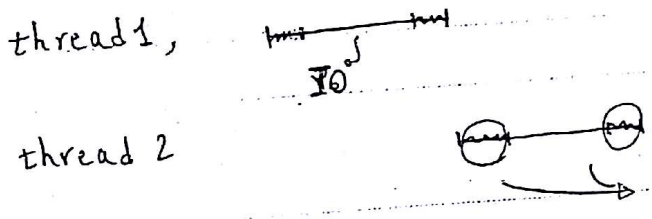
Subject
Date

Multi-Thread ها ~ در مقابل قبل قرار دارد و سبک وزن تر است و OS، نیاز به سربارهای زیادی ندارد.



یک تابع ؟ در main ؟ ایجاد ۲ thread نیاز داریم ؟ main آن ۲ را ایجاد می کند و خودش هم از بین می رود (در حالی که قبل از kill می شد)

" نسبت به قبلی تفاوتش در فضای آدرس دهی اشتراکی است " ∈ ؟ از لحاظ خوب است و از لحاظ غلط مثل قبل بد است.



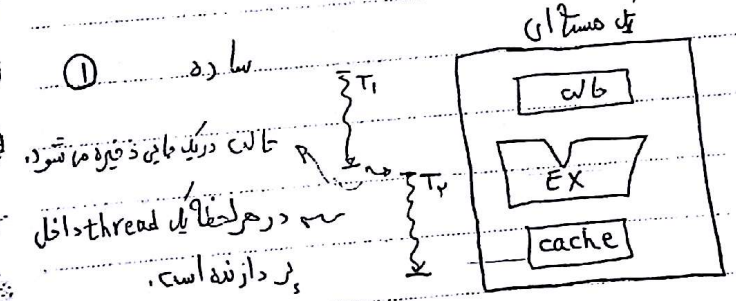
برای ذخیره/لود کردن حالت ترد در پردازنده

= مقادیر متغیرهای داخل رجیسترها

رجیسترهای خاص PC و SP و ...

وقتی cpu از thread گرفته می شود ؟ یا کارآن تمام شده یا IO آمده است.

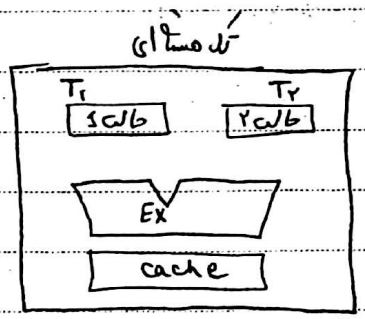
① ساده



حالت در یک جایی ذخیره می شود، سپس در هر لحظه یک thread داخل پردازنده است.

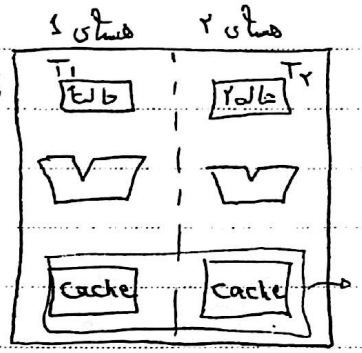
Subject
Date

② Hardware MultiThreading (Hyper Threading)



تفاوت این با قبل در این است که در قبل پس از انجام T_1، پردازنده متوقف می‌شود تا OS، T_1 را بگیرد و T_2 را تحویل دهد ولی در این جا این توقف را نداریم چون پس از اجرای T_1، با صورت سخت افزاری با T_2، سوئیچ می‌کنیم و OS در همین حال T_1 را عوض می‌کند و با همین ترتیب ادامه می‌دهد ولی هم روند است تا هم زمان چون در هر لحظه فقط یک می‌تواند اجرا شود چون فقط یک واحد محاسباتی داریم.

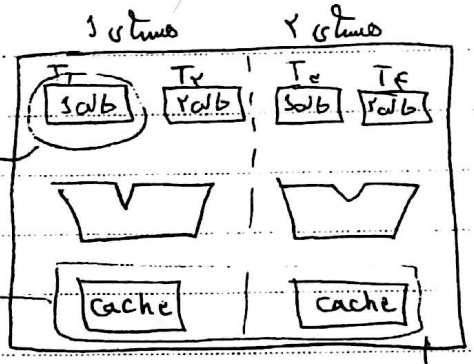
③ Multi-core



T_1 و T_2 با صورت هم زمان اجرا می‌شوند

④

ترکیب ۲ و ۳



۴ ترد هم زمان در پردازنده هستند و ۲ تا با صورت هم زمان اجرا می‌شوند

شکل ④ ام

این خوب است چون اگر یک هسته با هر دلیل خراب شد cache خود را با هسته ۲ می‌دهد و عملکرد و بازدهی آن را بالا نمی‌برد چون cache آن برگشته است.

core i7 ← ۴ هسته و داخل هر هسته ۲ ترد

---

① coarde grainned ← مشکل برخورد swip می‌کند

② fine ← با صورت متناوب switch کند

پایان Multi core

Subject
Date

برای دستگاه‌های I/O ۳ تا امکان پذیر است.

① polling سرکشی ... من وند تا ببیند درخواست دارد و اطلاعاتی ...

② Interrupt وقفه ... Device یا I/O درخواستی ... cpu را ... می‌کند.

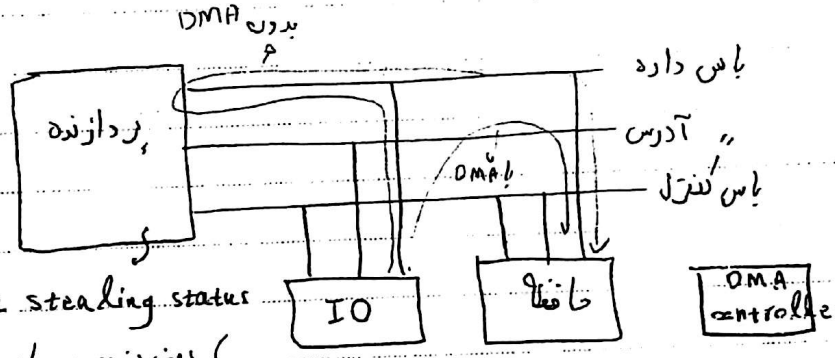
③ Direct Memory Access (DMA) ⇒

حالتی است که می‌خواهیم حجم زیادی را بین حافظه و I/O مبادله کنیم.

(در پردازنده در loop انتقال است)

پس وقتی زیادی از پردازنده سر این مورد تلف می‌شود.

بنابراین یک دستگاه ارزان قیمت با نام DMA controller قرار می‌دهیم و پردازنده را از این وسط کنار می‌گذاریم و پردازنده فقط DMA کنترل را تنظیم می‌کند و وظیفه انتقال بر عهده DMA است.



یعنی در دیدن باس

پردازنده وقتی اطلاعات بین حافظه و IO منتقل می‌شود باس کار می‌کند و می‌تواند از آن استفاده کند یا زهم پردازنده متوقف می‌شود ولی در این حالت کار پردازنده را انجام می‌دهد و وقتی این وسط کار نداشت یا IO کار می‌کند.

۹۲/۱۰/۷

مثال) * پردازنده با فرکانس کاری : ۵۰۰ MHz

این زمان سربار تحمیل می‌کند.

* سرکشی : ۲۰۰ طول می‌کشد برنامه‌ی سرکشی مربوط به Polling

Subject
Date

* ① Mouse : ۳۰ بار سرکشی در ثانیه

* ② Floppy Disk : واحدهای داده‌ای ۱۶ بیتی با نرخ انتقال ۵۰ kB/sec

* ③ HDD : ۴ MB/sec ، بلاک‌های ۴ کیلوبایتی

فرض ⇒ CPI = ۱۰۰ متوسط زمان هر دستور

① $۳۰ * ۴۰۰^{c.c.} = ۱۲,۰۰۰^{c.c.}$ ⇒ سربار $= \frac{۱۲,۰۰۰}{۵۰۰ \times ۱۰^۶} = ۰/۰۰۲٪$

این زیاد مهم نیست یعنی

در هر ثانیه باید ۲۵ هزار بار سرکشی کنیم تا هیچ داده‌ای را از دست ندهیم

این میزان از زمان CPU تلف می‌شود.

② $\frac{۵۰ kB/sec}{۲ B/polling} = ۲۵k \text{ polling/sec}$

سربار $= \frac{۲۵^k \times ۴۰۰}{۵۰۰ \times ۱۰^۶} = ۲٪$ ⇒ این هم تا حدی قابل قبول است.

② سربار $= \frac{\frac{۴ MB/sec}{۱۶ B/polling} \times ۴۰۰}{۵۰۰ \times ۱۰^۶} = ۲۰٪$ ⇒ این دیگه واقعاً قابل تحمل نیست.

۲۰٪ زمان CPU را تلف می‌کند.

لذا بنابراین Polling برای device های کند مناسب است چون با یک برنامه‌ی ساده امکان‌پذیر است ولی device های با حجم اطلاعات انتقالی آن‌ها زیاد است سرکشی مناسب نیست.

روش وقفه: برای سرویس وقفه $۵۰۰^{c.c.}$ برای هر انتقال نیمه (هارددیسک در ۵٪ موارد داده‌ای برای ارسال دارد.)

سیکل‌های درثانیه برای هارددیسک $= ۲۵۰ \times ۵۰۰ = ۱۲۵ \times ۱۰^۶$ cycle per second

در روش قبل ما باید همیشه سرکشی را انجام دهیم چون نمی‌دانیم device ی اطلاعات را می‌فرستد یا خیر ولی device ها خیلی سریع است.

ولی در روش وقفه فقط زمانی که می‌خواهیم اطلاعات را انتقال دهیم به سراغ آن می‌رویم.

سربار $= \frac{۵٪ \times ۱۲۵ \times ۱۰^۶}{۵۰۰ \times ۱۰^۶} = ۱/۲۵٪$ ⇒ این درصد قابل قبول است.

لذا بنابراین اگر حجم اطلاعات زیاد باشد روش وقفه از روش سرکشی خیلی بهتر است.

Subject
Date

روش DMA :

* تنظیم DMA در شروع کار: ۱۰۰۰ سیکل
* وقفه در انتهای انتقال: ۵۰۰ سیکل
* هارددیسک: $4MB/sec$ سه ۱۰۰٪ زمان درحال ارسال اطلاعات ⇐ بدترین حالت ممکن
* هر انتقال DMA : ۸kB (۸kB داده را جابجا می‌کنیم)

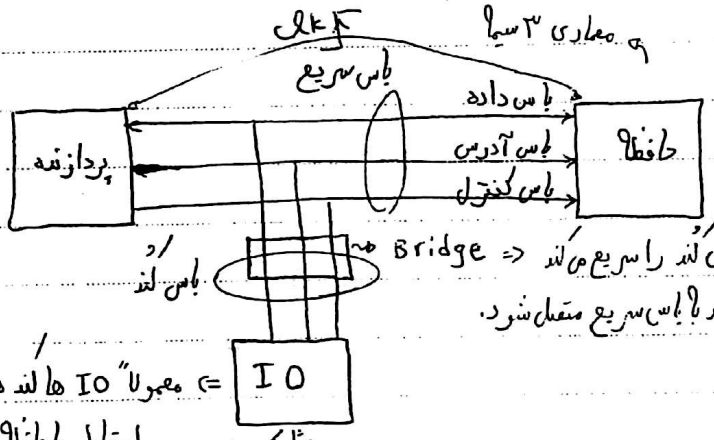
$$\frac{8kB}{4MB/sec} = 2\times10^{-3}\ second$$

$$\frac{(1000+500)\ cycle/Transfer}{2\times10^{-3}\ sec/Transfer} = 750\times10^{3}\ cycle/sec$$

تازه ما فرض کردیم این بدترین حالت باشه ⇐ ٪ خیلی مناسب است $\frac{750\times10^{3}}{500\times10^{6}} = 0.2\%$: سربار

درطول ٪ هارددیسک در ۱۰۰٪ موارد انتقال اطلاعات انجام نمی‌دهد.

انواع گذرگاه‌ها (Bus) : معماری ۳ سیما ⇐



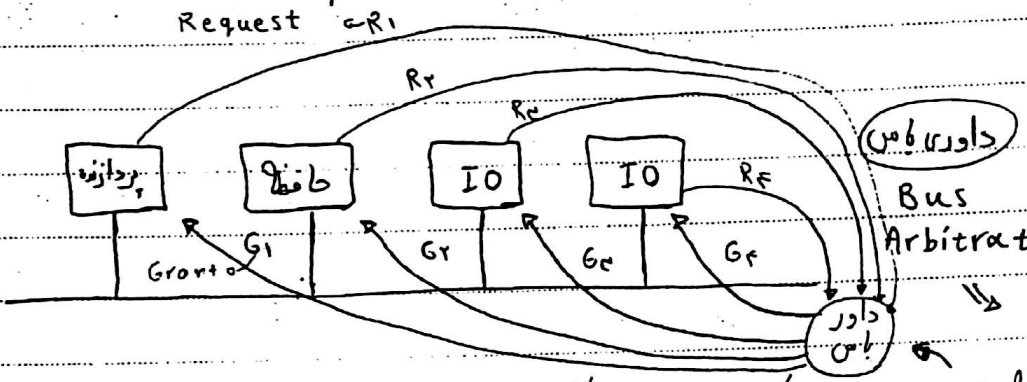
این وسیله، باس کند را سریع می‌کند ⇐ و می‌تواند با باس سریع متصل شود.

معمولاً IO ها کند هستند ولی باس ارتباطی با حافظه خیلی سریع است. ⇐ مثل کیبورد...

Subject
Date

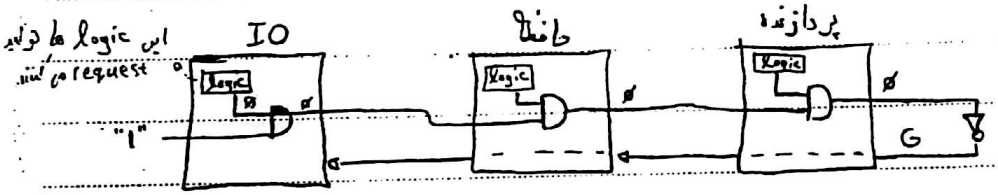
هر استاندارد باس یک timing مخصوص به خود دارد برای گذاشتن داده و ...

که می خواهیم مکانیزم دسترسی چند وسیله به یک باس را بررسی کنیم.



* centralized : یک داور باس داریم که R ها به آن می آیند و بر اساس اولویت ها باس را به یکی وسیله اختصاص می دهد. ← یک مشکل آن این است که اگر وسیله ها زیاد شود طراحی داوری باس مشکل می شود.

* Distributed : ← Daisy chain.



← موقعیت فیزیکی این device ها از وسیله ای که داوری را انجام می دهد اولویت را مشخص می کند.

Req = تولید

دراین جا پردازنده با اولویت اول وسپس حافظه وسپس IO چون Grant اول به پردازنده می رسد و اگر پردازنده Req داده باشد باس را در اختیار می گیرد وگرنه آن را به حافظه پاس می دهد و همین طور تا انتها می رود.

پس در این جا داوری به صورت متمرکز نیست و همه ی وسایل با هم داوری را انجام می دهند و Distributed داور انجام می شود و وسیله ی داوری هم بسیار ساده است (مثلاً در این جا یک گیت Not)

PAPCO

(۴

Subject
Date

مثال) سنکرون — آسنکرون

clk : 50^{ns} — هر Handshake : 40^{ns}

هر انتقال روی باس یک سیکل

حافظه : $t_{Acc} = 200^{ns}$ ؛ ۳۲ بیت اطلاعات در هر دسترسی جابجا می‌شود.

می‌خواهیم اتصال حافظه را با این ۲ باس بررسی و مقایسه کنیم:

سنکرون : ارسال آدرس + تأخیر حافظه + ارسال داده

$50^{ns} + 200^{ns} + 50^{ns} = 300^{ns}$

300^{ns} — 4^{byte} ، 1^{sec} — x } BW = 13.3 MB/sec

آسنکرون : طبق شکل قبل

مرحله ۱ : 40^{ns}

مرحله ۲ و ۳ و ۴ : $3 \times 40^{ns} = 120^{ns}$ } $max(120, 200^{ns}) = 200^{ns}$

همزمان با دسترسی به حافظه ، 200 ns

گفتیم از اول Ack اول را نیاز دارد اطلاعات دارد انجام می‌شود با صورت موازی با Handsh

مرحله ۵ و ۶ و ۷ : $3 \times 40^{ns} = 120^{ns}$

مجموع $= 40 + 200 + 120 = 360 ns$

360 ns — 4 b ، 1^{sec} — x } ⇐ BW = 11.1 MB/sec

که خیلی بدتر نیست.

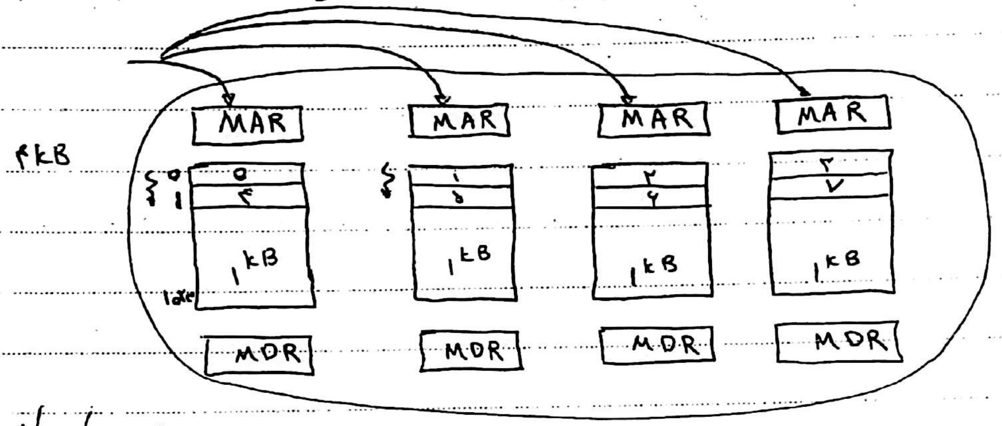
⇐ راه‌های افزایش BW : ① جداسازی باس آدرس و داده ② افزایش حجم انتقالی (۳۲ بیت را افزایش دهیم).

که چیزی که معمولاً استفاده می‌شود انتقال بلوکی است.

Subject
Date

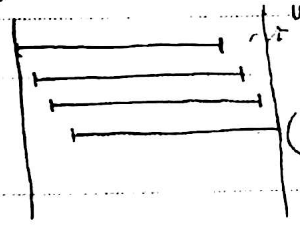
Block Transfer: این می‌گوید اگر می‌خواهی با چند خانه‌ی متوالی حافظه دسترسی پیدا کنی برای هر بار به اندازه‌ی اول نیاز نیست زمان صرف کنی.

→ Memory Interleaving



۴ انتقال در یک سیکل (تقریباً)

این روش از ۲ روش قبل عملی‌تر است. ⇐



از این زمان‌های کوچک می‌توان صرف نظر نمود

و فرض کرد که با یک سیگنال خواندن ۴ کلمه به طور همزمان آماده می‌شوند و انتقال می‌یابند

مثال از روش بالا:

* انتقال بلوک‌های ۴ تا ۱۶ کلمه‌ی ۳۲ بیتی

* باس ۶۴ بیتی با فرکانس 200MHz ($5ns$)، هر انتقال ۶۴ بیتی ۱ سیکل، ۱ سیکل برای ارسال آدرس به حافظه

* بین هر ۲ عملیات باس ۲ سیکل فاصله

$4 \times 5^{ns} = 20^{ns}$

این overlap با ۲ تا ۲ c.c. دارد و مشکل ندارد ما

* حافظه با زمان دسترسی 200^{ns} برای ۴ کلمه اول و 20^{ns} برای دسترسی‌های بعدی

ولی اگر یک نبود باید را هم man کنیم

* می‌خواهیم ۲۵۶ کلمه را از حافظه بخوانیم، $BW = ?$

Subject
Date

① بلوک های ۴ تایی

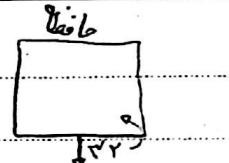
* برای ارسال آدرس $1^{c.c.}$
* $\frac{200^{ns}}{5^{ns}} = 40^{c.c.}$
* برای قرار دادن داده روی باس $2^{c.c.}$
* $2^{c.c.} \rightarrow idle$

$\rightarrow + 48^{c.c.}$

کلاک ها را با صورت بسته ۴ تایی می فرستیم

$* \frac{256}{4} = 2880^{c.c.}$ (trans. 9)

$BW = 4.44^{MT/sec} \rightarrow = 71.11^{MB/sec}$



$* 5^{ns} =$ ◯ با تناسب نوشته

① بلوک ۱۶ تایی

* برای ارسال آدرس : $1^{c.c.}$
* برای دسترسی حافظه : $200^{ns} = 40^{c.c.}$
* برای ارسال داده روی باس : $2^{c.c.}$
* idle بین دو عملیات باس : $2^{c.c.}$

$41^{c.c.} + 4(2+2) = 57^{c.c.}$

$57^{c.c.} * \frac{256}{16} = 912^{c.c.}$

نمودار انتقال

$BW = 3.81^{MT/sec} = 224.56^{MB/sec}$

این مهم است.