Subject:

Year. Month. Date. ( )

معماری کامپوتر

دکتر محمدرضا رزنکی

Subject.

Year. Month. Date. ( )

سخت افزار کامپیوتر

مفاهیم مورد نیاز از درس مدار منطقی:

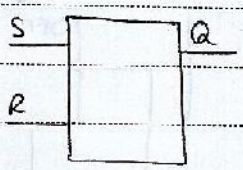
FF ، RAM ، ROM ، MUX ، DeMux ، Decoder ، encoder ، PROM

EPROM ، E²PROM ، SRAM ، DRAM ، Tri state ، BUS

FF:

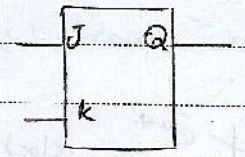
1) SR:

S	R	Q_0 (اولی)	Q_1 (بعدی)
0	0		Q_0
0	1		0
1	0		1
1	1		?



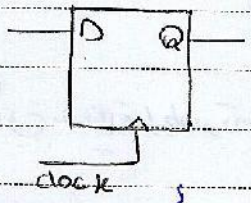
2) Jk:

J	k	Q_0	Q_1
0	0		Q_0
0	1		0
1	0		1
1	1		$\overline{Q_0}$



3) D FF:

D	Q
0	0
1	1



در زمانی که پالس بالارونده تولید D منتقل می‌شود

4) T FF:

T	Q_0	Q_1 (بعدی)
0		Q_0
1		$\overline{Q_0}$

این نوع FF برای ساختن clock مناسب است.

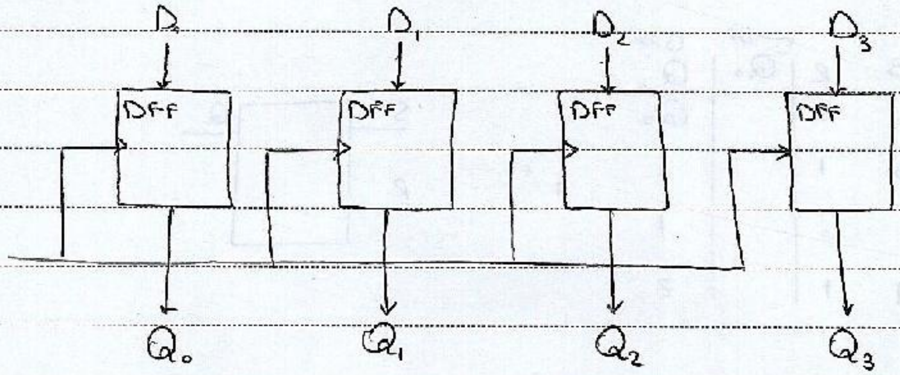
paterson ← performance ، piple

E ، programming, IO ، 5

همه FF ها می‌توانند حساس به لبه باشند و یا دارای clock، مهم‌ترین آنها DFF است.

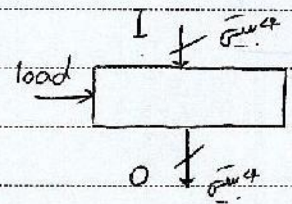
Register :

مجموعه‌ای از FF ها کنار هم register را می‌سازد.



با روی هر لبه مقادیر ورودی D_0 تا D_3 منتقل می‌شود.

از آنجایی که نمی‌خواهیم در هر clock اطلاعات منتقل شود از enable استفاده می‌کنیم برای اینکه data از اعتبار برخوردار باشد.



در لحظه clock

$$\begin{cases} \text{if } load = 1 : O = I = Q \\ \text{if } load = 0 : O = Q \text{ قبلی} \end{cases}$$

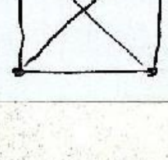
⚠ : هیچ گاه نباید پایه load خالی باشد حتی اگر احتیاج به آن نداریم قرار داده و مقدار آن را 1 قرار می‌دهیم.

برای اتصال دو رجیستر به یکدیگر از چندین راه می‌توان استفاده کرد :

1- point to point

هر نقطه را به سایر رجیسترها وصل کنیم ولی این راه از لحاظ سادگی مناسب است اما دارای اشکال های زیر می‌باشد :

1- سیم‌های زیادی مصرف می‌شود 2- گره‌های زیادی ایجاد می‌کند 3- قطع و وصل شدن سیم‌ها

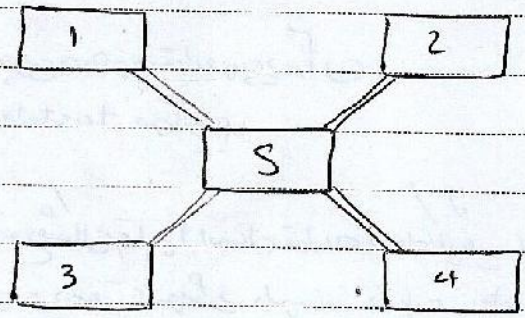


Subject:
Year. Month. Date. ( )

۲_ Star :

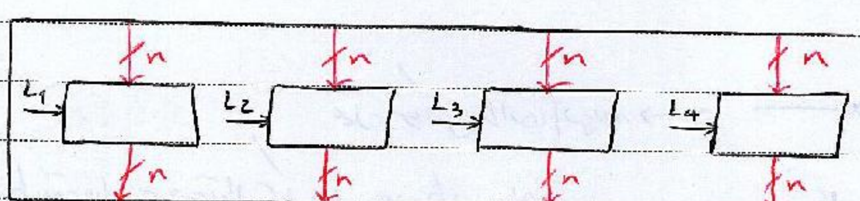
انتقال بین دو رجیستر از طریق سوئیچ‌ها انجام می‌شود و برای انتقال دو شاخه سیگنال باید انتخاب star فرستاده شود و آن داده را انتقال می‌دهند ولی در روش قبلی هر دو گره مستقیماً می‌توانستند داده ارسال کنند.

در این روش هم‌زمانی ارتباط وجود ندارد و کارایی پایین است.



۳_ BUS (گذرگاه) :

بهترین روش برای انتقال جهت انتقال داده است. از نظر مصرف توان و حرارت بهترین مصرف را دارد. load ها اجازه می‌دهد داده را به کدام رجیستر وارد کنیم



بسته به فعال بودن هر یک از رجیسترها مقدار روی BUS درون رجیستر مربوط ذخیره می‌شود.

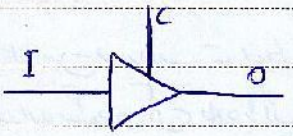
⚠ هیچگاه سیگنال‌های مقدار دهنده را مستقیماً به سیگنال متصل نکنید. برای رفع اشکال از MUX استفاده می‌کنیم. در واقع همه سیگنال‌های تزریق کننده مقدار به BUS توسط MUX از BUS جدا می‌شوند.

در واقع ورودی توسط enable و خروجی از طریق MUX کنترل می‌شود.

Subject:

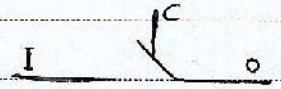
Year. Month. Date. ( )

Tri_state



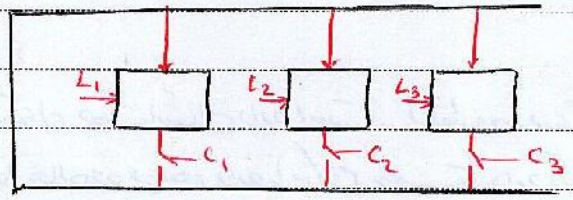
$$O = \begin{cases} I & \text{if } C = 1 \\ Z & \text{if } C = 0 \end{cases}$$

high_impedance

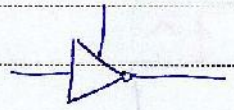


$$O = \begin{cases} 0 \\ 1 \\ 2 \end{cases} \rightsquigarrow$$ چون خروجی سه مقدار می‌پذیرد به آن tristate می‌گوییم.

برای رفع مشکل قبلی (اتصال چند خروجی به یکدیگر) می‌توان از tri state استفاده کرد که براساس جدول زیر درست می‌شود و می‌تواند دو یا سه tristate هم‌زمان در مدار باشد.

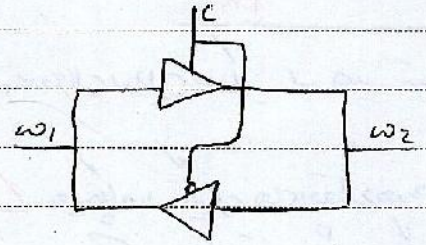


Z	0	0
Z	1	1

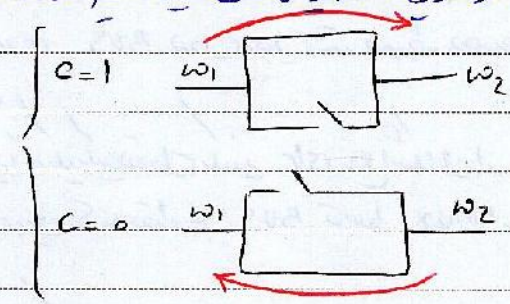


عمل بافر را انجام می‌دهد.

نکته: از طریق مدار زیر می‌توان یک سیم دو طرفه ساخت.



$\Rightarrow$



Subject:

Year. Month. Date. ( )

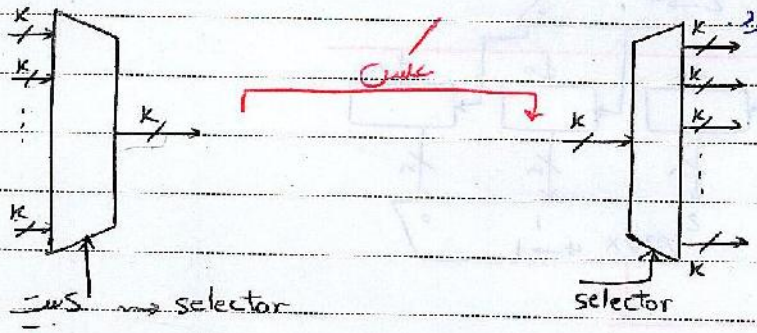
سطح تجرید

پایین ترین سطح تجرید گیت و ترانزیستور است.

سخت افزار	نرم افزار	
Chip Level	(Behavior level)	→ IC ، BUS ، ...
RTL (Register transfer level)		→ register ، mux ، Decoder ، ...
gate abstraction level		→ AND ، OR ، not ، FF ، ...
transistor Abstraction Level		→ ترانزیستور سلف ، مقاومت ، خازن

MUX (انتخاب کننده)

باید بتوانی ورودی و خروجی یا کدیگر را برابر باشند و براساس مقدار selector مشخص می‌شود که مقدار کدام پایه ورودی باید منتقل شود.

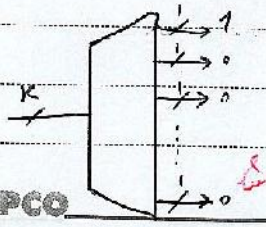


MUX

DeMUX

DeMUX: مشخص می‌کند که ورودی روی کدام پایه خروجی قرار گیرد و سایر پایه‌ها که مقدار ندارند high impedance می‌شوند.

Decoder (رمزگشا)



هیچ پایه selector ندارد. پایه‌ها همیشه با ورودی مقدار یک و سایر پایه‌ها مقدار صفر را می‌گیرند. در هر لحظه فقط و فقط یک پین خروجی فعال است.

فقط یک پین خروجی فعال در هر لحظه ← اگر دقیقا همین باشد

هر لحظه یکی از خروجی‌ها صفر و بقیه "1" ← اگر دقیقا همین باشد

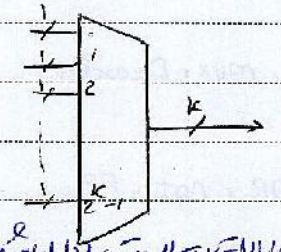
PaPCO

Subject.

Year. Month. Date. ( )

Encoder

selector برعکس دیکدر فقط یکی از پایه‌ها فعال است و شماره‌ی پایه‌ی فعال در خروجی ظاهر می‌شود. ولی این نوع encoder هیچ کنترلی روی فعال بودن تمامی پایه‌ها ندارد برای همین از نوع دوم استفاده می‌کنیم.

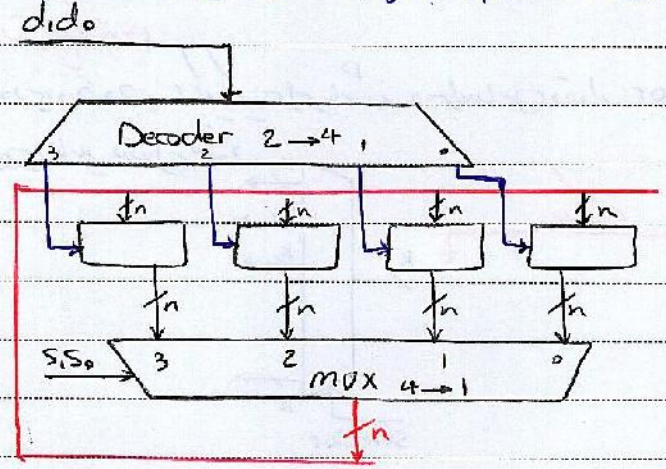


(رمزگذار الویت دار) priority encoder

می‌تواند تعدادی از ورودی‌ها فعال باشند شماره‌ی ورودی انتخاب می‌شود ۱- فعال باشد

۲- بین فعال‌ها بالاترین الویت را دارا باشد

نکته: اگر همه‌ی ورودی‌ها غیرفعال باشند خروجی بستگی به ساختمان آن دارد.



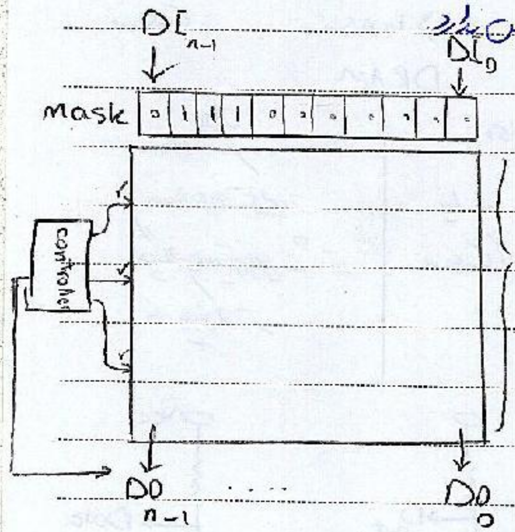
$(d_1, d_0, S_1, S_0) \longrightarrow (0, 1, 1, 0) \equiv R_1 \leftarrow R_2$

Memory
- RAM (random Access Memory) نقطه‌ای / قابل sequential Access Memory
- ROM (Read only memory)

Subject:
Year. Month. Date. ( )

حافظه های sequential برای دسترسی به یک داده باید از ابتدا بر داده ها عبور کنیم در نتیجه سرعت دسترسی به آن ها کم است. دسترسی در RAM دو طرفه است ولی در ROM تنها می توان اطلاعات موجود درون آن را خواند و دسترسی یک طرفه است.

دسته دیگری از حافظه ها CAM است (Content Adderessable Memory) که آدرس دهی آن بر اساس محتوا است و برای نوشتن یا خواندن یک داده نیاز به داشتن آدرس ندارد.



داده ها از طریق Mask وارد می شوند و به ازای بیت هایی که یک هستند وارد حافظه می شود و به دنبال الگویی می گردد که با این ها match شود. الگو با mask باید به گونه ای انتخاب شود که تنها یک داده match شود و در صورت داشتن چندین کاندید از طریق controller و بر اساس طراحی یکی انتخاب می شود.

در هنگام استفاده از یک سخت افزار نکات زیر اهمیت دارد:

1. Area: هزینه طراحی یا HW overhead یا هزینه سخت افزاری
2. performance: باید کارایی بالایی داشته باشد البته با توجه به سایر هزینه ها
3. Timing: داشتن سرعت بالا
4. power: توان مصرفی پایین.

نکته: در طراحی های یک سخت افزار هرچه نسبت زیر بیشتر باشد سخت افزار بهتری خواهیم داشت

$$\text{اندازه} = \frac{\text{Performance}}{\text{cost}}$$

در مورد حافظه CAM می توان گفت performance بالایی دارد ولی Area زیادی مصرف می کند و عملاً دیگر از آن استفاده نمی کنند.

$$Area_{CAM} = \sqrt{2}\, Area_{RAM}$$

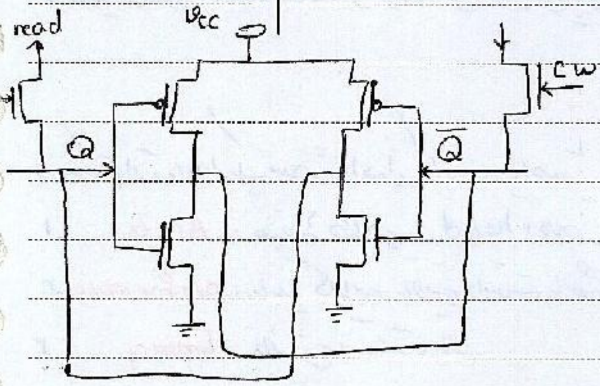
Subject.
Year. Month. Date. ( )

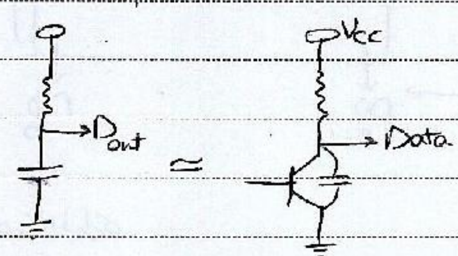
حافظه‌ها از نوع RAM به دو دسته تقسیم می‌شوند

RAM:
- SRAM (static RAM): از نظر سرعت کمتر از DRAM است. feedback دارد در صورت قطع شدن برق می‌تواند خودش تغذیه کند ولی در نهایت با قطع برق داده از بین می‌رود
- DRAM (Dynamic RAM): این نوع حافظه Area کم‌تری نسبت به SRAM در بر می‌گیرد.

سرعت: DRAM $\simeq > 10ns$ — SRAM $\simeq < 10ns$

DRAM مزایا	DRAM معایب	SRAM مزایا	SRAM معایب
مساحت کمتر	نیاز به refresh دارد	سرعت بالا	مساحت بیشتر
چگالی بیتی بیشتر	سرعت کمتر نسبت به SRAM	توان کم	هزینه بالا
هزینه کمتر		نیاز به refresh ندارد	



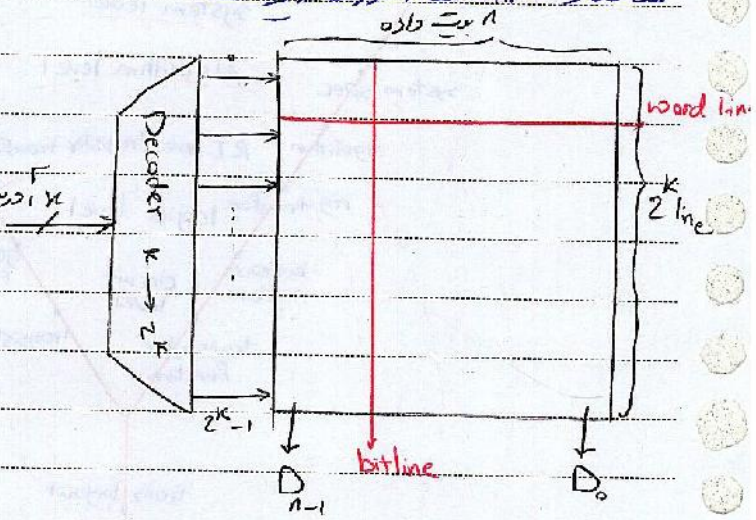


DRAM:
- SDRAM (Synchrouneus DRAM): clock بیرونی را به آن اضافه کردند در نتیجه عمل refresh نیز انجام می‌شود
- DDR RAM (Double Data Rat DRAM): لبه‌ی بالا رونده و پایین رونده هر دو clock می‌زنند در نتیجه سرعت بالایی دارد و تقریباً سرعت دو برابر می‌شود.

Subject:

Year. Month. Date. ( )

ساختار RAM ها به صورت زیر است:



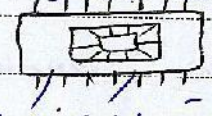
تفاوت SRAM و DRAM در ساختار حافظه برای ذخیره داشتن اطلاعات است که براساس line انتخاب شده خروجی آن bitline متصل می‌شود. برای بالا بردن کارایی باید word line کوتاه کنیم و همچنین استفاده از decoder مجبور باید.

ROM ← ROM

ROM: (read only memory) تنها قابلیت خواندن دارد و یک بار مصرف است زیرا طرز ساخت آن با سوزاندن ترانزیستور تکمیل می‌شود. کاربرد آن اطلاعات در CMOS یا مشخصات و تنظیمات کارخانه را شامل است.

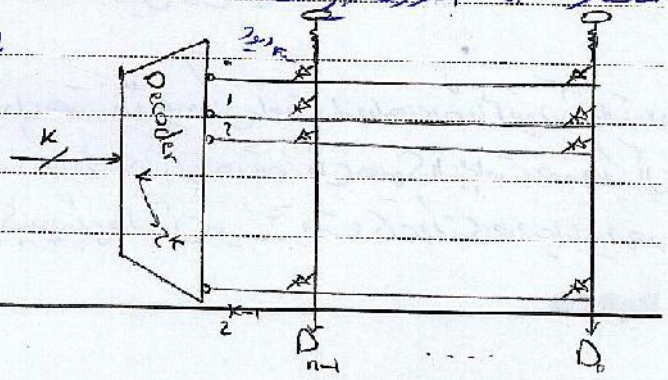
PROM ← PROM: (programable ROM) جنس آن از نوع فیوز است و تنها یک بار می‌توان روی آن اطلاعات را نوشت و بعد از آن تنها قابل خواندن است.

EPROM ← EPROM: (Ereasable PROM) بعد از program کردن می‌توان اطلاعات آن را پاک کرد. پاک کردن آن از طریق اشعه فرابنفش دوست ۱۵ دقیقه است. بعد از program شدن دوست نیاز به پاک کردن باید قطعی را جدا کرد و به مدت پاک کرد.

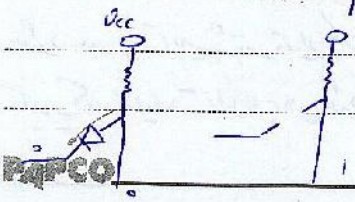


EEPROM ← EEPROM: (electrical EPROM) پاک کردن آن نیز از طریق مدار الکترونیکی صورت می‌گیرد و در حذف EPROM می‌توان قسمتی از اطلاعات آن را پاک کرد. کاربرد آن در Flash هاست.

ساختار ROM به صورت زیر است:

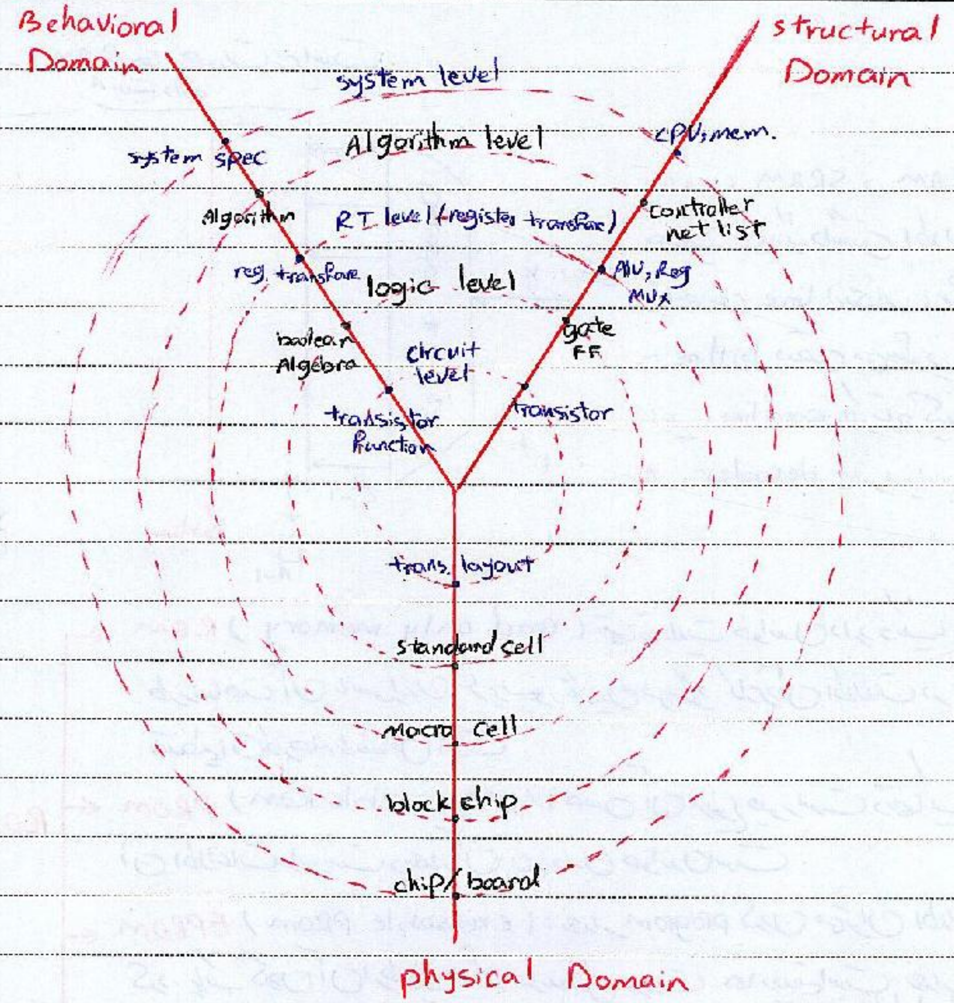


با سوزاندن دیود ها می‌توان اطلاعات را در روی ROM وضیو کنیم



Subject:
Year. Month. Date. ( )

" Y chart "



مدارها به دو دسته synchroneous (همگام) و Asynchronous (ناهمگام) تقسیم می‌شوند:

۱- synchronous circuit : این نوع مدارها با وجود عدم مناسب بودن ولی به خاطر طراحی ساده‌تر کاربرد بیشتری دارند. برحسب clock عملیات انجام می‌شود که سرعت کمتری نسبت به آسنکرون دارد زیرا clock در سطح آن پخش می‌شود. همین clock ها باعث کاهش سرعت و افزایش توان می‌شود.

۲- Asynchronous circuit : بهتر از دسته اول هستند چون ممکن است از مدارهای همگام نیز در آن استفاده شود. سرعت بیشتر، توان کمتری دارد، در اینجا عملیات کمتری انجام می‌دهد تنها آن هایی که لازم است هنگامی که آن تغییر کند عملیات اعمال می‌شود. ولی از آن جایی که پیچیدگی مدار آن بیشتر است کاربرد کمتری دارد زیرا باید در هم

PAPCO

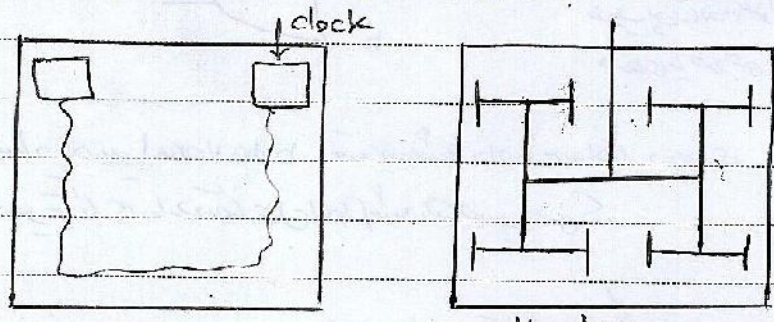
Subject: J. BAHSKAR, A verilog HDL primer, star galaxy press, 1997

Year. Month. Date. ( )

لحظه داده ها را پردازش کند.

نکته ۵: می توان مدارهای ناهمگام را با اضافه کردن پایه clock به همگام تبدیل کرد ولی عکس آن امکان پذیر نیست.

مشکل مدارهای همگام این است که لبه بالا رونده برای همه قسمت های یک مدار در زمان یکسان اتفاق نمی افتد به این مشکل clock skew می گویند. برای حل این مشکل از H_tree استفاده می کنند تا clock در زمان یکسان به همه قسمت های مدار برسد. البته مشکل به طور کامل حل نمی شود و در بعضی موارد بهبود ده clock پخش می شود.



H_tree

HDL ← Hardware Design Language

این زبان دیگر کاربرد ندارد به جای آن از زبان های زیر استفاده می کنند:

۱_ VHDL: جزئیاتی از سخت افزار در آن دیده نمی شود و بسیار خلاصه است و چون خیلی عمومی صحبت می کند کار کردن با آن دشوار است

۲_ verilog: استانداردهایی ایجاد شدند که از آن LRM (language resource manual) خیلی بهتری برای استفاده کنندگان ایجاد شد.

۳_ systemC: همان کاربرد آن را می شود ولی از verilog سخت افزاری تر است و از زبان C++ در آن استفاده شده است.

برنامه ای که توسط این زبان ها نوشته می شود باید شبیه سازی شود که برای این کار از modelsim استفاده می کنند. یک شبیه ساز همزمانی است که می تواند هر قسمت از برنامه را جدا از برنامه های بالا نوشته شده است.

Subject:
Year. Month. Date. ( )

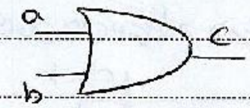
: module

یک طرح سخت افزاری دیجیتالی از module هاست که باید با یکدیگر در ارتباط هستند کوچکترین component قابل توصیف را module می گویند

به جز خط اول و دوم بین خط ها فرق دارد.

```
module اسم (port list )
            c, a, b
```
(پورت ورودی ؛ پورت خروجی)

```
output c;
input a,b;
```
} تعریف پورت ها به عنوان ورودی یا خروجی




```
reg c;
always a, b
begin c=a | b;
end
end module
```

این سطح تجرید behavioral استفاده شده و اصلاً به ساختار و معماری کامپیوتر کاری ندارد و تنها عمل انجام شده را توصیف می کند.

چون می خواهد مقداری در خودش نگه دارد باید به عنوان reg نیز تعریف شود. output و input هم جزو سیم ها هستند.

```
module اسم( port list )
  reg , wire, parameter,
  input, output, inout,
  function, task, ...
  initial statement
  Always "
  module instaniation
  gate "
  UDP "
  continous assignment
end module
```

Declarations: reg, wire, parameter, input, output, inout (هم ورودی و هم خروجی), function, task, ...

(function → مقداری را return می کند)

Statements: initial statement, Always, module instaniation, gate, UDP, continous assignment

سطوح تجرید در verilog:

- behavioral → initial statement / Always
- Data flow (RTL) → continous assignment
- gate level (gate → gate
- switch level (Cirant)
- structural level → module instaniation

Subject: زمانی که می‌خواهیم دستورات به ترتیب نوشته شده اجرا شود از begin - end استفاده می‌کنیم

Year. Month. Date. ( )

نکته: ترتیب نوشتن دستورها در module اهمیتی ندارد ولی هنگامی که بین begin و end نوشته شود ترتیب مهم است.

تعریف OR در سطح RTL:

```
module اسم (c,a,b)
    output c;
    input a,b;
    assign c=a|b;
end module
```

assign ← متغیری را که اختصاص می‌دهد باید از نوع wire باشد.

تعریف OR در سطح gate:

از جمله gate های موجود در verilog عبارتند از:

and ، or ، nor ، nand ، XOR ، Xnor ، not ، bufif ، nmos ، pmos ، cmos ، ....

```
module اسم (c,a,b)
    output c;
    input a,b;

    Or myOr(c,a,b);
end module;
```

Or myOr(c,a,b); ← باید برای gate نامی انتخاب کنیم و ورودی و خروجی موردنیاز آن را به عنوان ورودی به gate بدهیم.

نکته: module ها نباید درون یکدیگر تعریف شوند ولی درون یک module می‌توان module دیگری را صدا زد.

Subject:

Year. Month. Date. ( )

مثال:

```
module X(c,a,b)
    input a,b;
    output c;
    or myor(c,a,b);
endmodule

module Y(c,a,b,d)
    input a,b,d;
    output c;

    X myX(w1,a,b)
    X yourX(w2,b,d)
    X ourX(c,w1,w2)
endmodule
```

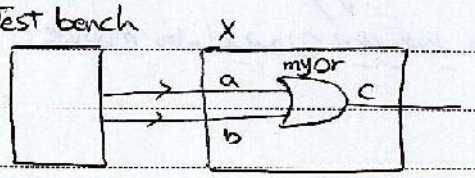
نکته: در زبان verilog wire یک مقدار 0، 1، X، Z را داشت باشد که X یعنی سیمی که wire هم صفر و هم یک ریخته شده است و Z زمانی که هیچ مقداری به wire داده نشده است که نشان wire است ولی برای X، register نیز می‌تواند همین مقداری را داشته باشد.

در نتیجه برای تست module نوشته شده باید مقداردهی اولیه بکنیم که برای اینکار از test bench استفاده می‌کنیم که می‌توانیم module باشد که فقط مقداردهی می‌کند.

برای مقداردهی کردن از دو روش می‌توان استفاده کرد:

۱- استفاده از خود modelsim که این قابلیت را دارد که در هنگام ورودی wire مقداری را اختصاص دهیم.

۲- استفاده از test bench:



initial را زمانی که بخواهیم متغیرها را مقداردهی اولیه کنیم استفاده می کنیم.

Test Bench تماماً خروجی دارد — باید یک instance از ماژول درون آن بنویسیم.

```
module TestBinch (a,b)
  output a,b;
  reg a,b;
  initial
    begin
      a = 1;
      b = 0;
    end $finish;
end module
```

باید ورودی های ماژول را reg و خروجی را از نوع wire تعریف می کنیم

از آن جایی که output و input از نوع wire است در محدوده begin و end نمی توان مقداری به آن ها داد در نتیجه از reg استفاده می کنیم

⚠ انتسابات در دستورات initial و always فقط باید به reg انجام شود

برای شبیه سازی برنامه نیاز به topic module دارد تا هنگام شبیه سازی اجرا شود.

```
module project;
wire Z,r,S;
X   MX(Z,r,S);
Testbench  MT(r,S);
endmodule
```

چرا همه چیز را از نوع wire تعریف کردیم؟

برای ایجاد تاخیر درون test bench باید از دستور # استفاده کنیم تا بتوان بین ورودی ها دوباره مقداردهی آن را هم عوض کرد.

```
#5  a = 0;
#5  b = 1;
```

بعد از 5 واحد زمانی مقدار را عوض می کند

Subject:

Year. Month. Date. ( )

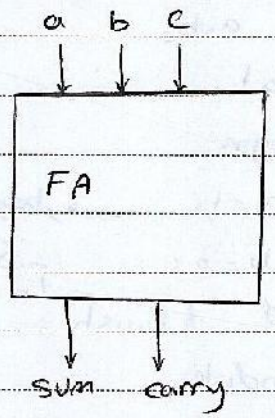
`timescale 1ns/100ps ; → برای مشخص کردن واحد زمانی از دستور زیر استفاده می کنیم

واحد زمان ↓ ↳ واحد برای رند کردن

مثال: طراحی fullADDer از طریق verilog.

سطح تحریک gate ↑

```
module FA(sum, carry, a, b, c)
  output sum, carry;
  input a, b, c;
  wire w1, y1, y2, y3, z1;
  xor xor1(w1, a, b);
  xor xor2(sum, w1, c);
  and and1(y1, a, b);
  and and2(y2, b, c);
  and and3(y3, c, a);
  or or1(z1, y1, y2);
  or or2(carry, z1, y3);
```



$sum = a \oplus b \oplus c$

$carry = ab + bc + ca$

سطح تحریک RTL :

```
assign sum = a ^ b ^ c;

assign carry = (a & b) | (b & c) | (c & a);
                  ↓ and    ↓ or
```

Subject:
Year. Month. Date. ( )

سطح behavior :

```
reg sum, carry;
always @ a, @b, @c
  begin
    sum = a ^ b ^ c;
    carry = a&b | b&c | c&a;
  end
```

نکته 8: می‌توان با داشتن جدول درستی مورد نظر gate جدیدی طراحی کرد. برای این کار از primitive استفاده می‌کنیم

```
primitive crctp(x, A, B, C)
  output x;
  input A, B, C;
  table
    000 : 1;
    001 : 0;
     :
  endtable
endprimitive
```

نکته 8: test bench باید درون initial نوشته شود. در ضمن یک بار اجرا می‌شود و با قرار دادن دستورات always و repeat می‌توان به ازای هر تغییر در متغیرها آن را اجرا کرد.

```
initial
  begin
    D = 3'b000;
    reapeat (8);
    #(10) D = D + 3'b001;
end
```

repeat (8); ← دستور تکرار به تعداد دفعات درون پرانتز (8) اجرا می‌شود که می‌توان بدنه‌ای برای آن منظور گرفت که شامل چنین دستوری باشد

PAPCO

Subject:

Year. Month. Date. ( )

برای نمایش خروجی test bench می‌توان از دستورات زیر استفاده کرد:

$display: → $display (format, argument list);

تغییرات را یک بار نمایش می‌دهد و بعد از آن به سر خط می‌آید.

$write: مانند است با این تفاوت که به سر خط نمی‌رود display

$monitor: در زمان‌های مختلف تغییر کند نمایش می‌دهد.

$time: در زمان اجرا مقادیر متغیرها را نمایش می‌دهد.

$finish: شبیه‌سازی تمام شد.

$display("time = %0d A=%B", $time, A);

علامت‌های اختصاصی که مقدارشان خروجی داده است

تمرین: با استفاده از دستورات زیر می‌توان clock موج‌نما را به‌صورت بوجود آورد:

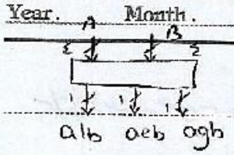
```
initial                          initial
begin                            begin
  clock = 1'b0;                    clock = 1'b0;
  repeat(30);                      #30 $finish;
  #10 clock = ~clock;            end
end                              always
                                   #10 clock = ~clock;
```

تابع initial یک بار اجرا شده ولی always همواره در حال اجراست.

* تمرین: هر یک از اجزای اولیه زیر را در سطح Gate با استفاده از Verilog (VHDL) توصیف نمایید.
هر توصیف باید همراه Test bench مربوط باشد که بتوان به راحتی توسط modelsim شبیه‌سازی کرد

Subject:
Year. Month. Date.

الف) Decoder 2→4
ب) Encoder 8→3
ج) MUX 8×3 → 1×3
د) Demux 1×3 → 8×3
ه) 4 bit comparator



1) net:

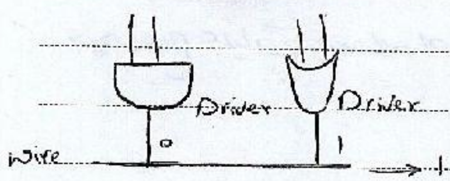
در Verilog keyword هایی که net هستند عبارت‌اند از:

wire ، tri ، wor ، wand ، trior ، triand

tri در Verilog دقیقاً برابر با wire است. (در خروجی این نوع تعریف می‌شود.)

در مدارها نمی‌توان دو خروجی را به یکدیگر وصل کرد ولی در Verilog گرچه این طرح درست نیست با طراحی‌های مختلف این کار امکان‌پذیر است

wor: خروجی‌ها را با یکدیگر or می‌کند و نتیجه را خارج می‌کند.



wand: در ساختار بالا خروجی‌ها را با هم and می‌کند.

trior و triand: مانند wor و wand عمل می‌کنند و تفاوتی ندارند.

2) reg:

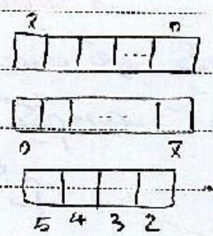
تنها برای نگه داشتن یک مقدار است که شامل حالت‌های زیر است:

reg x;

reg x[7:0]

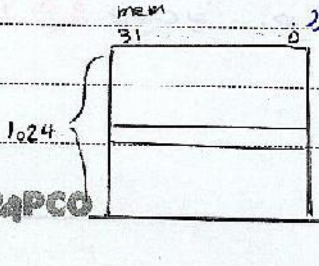
reg y[0:7]

reg t[5:2]



یک آرایه ۴ تایی ایجاد کردیم

می‌توان از این آرایه تایی به صورت عمودی به عنوان حافظه استفاده کرد



reg [0:1023] mem [31:0];

Subject:

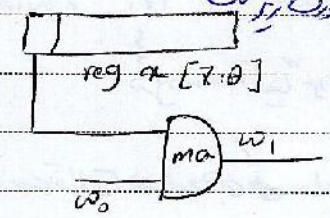
Year. Month. Date. ( )

برای دسترسی به بیت خاصی از یک word به این صورت تعریف می‌کنیم: reg a[31:0];

a = mem[5];

a[8]; بیت هشتم word ما

مثال: استفاده از یکی از طبقه‌های register سطح تجربه gate به صورت زیر است



and MA(w_1,w_0,a[7]);

مثال: می‌خواهیم عملیات حسابی اجرا شود داریم:

@ negedge ← لبه پایین رونده

always @ posedge clock ← لبه بالا رونده اجرا می‌شود.

begin

a = a+1;

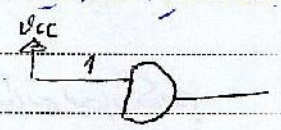
end

مثال: اگر بخواهیم از بازه‌ای از سیم‌های یک رجیستر آرایه‌ای استفاده کنیم می‌توان به صورت زیر عمل کرد:

a[12:8] = a[12:8] + 1

3) supply0 , supply1

زمانی که می‌خواهیم یک wire تعریف کنیم که Vcc وصل باشد و همیشه مقدار یک داشته باشد از supply1 استفاده می‌کنیم و در صورت داشتن مقدار صفر از supply0 استفاده می‌کنیم



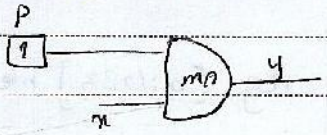
supply1 p;

مثال: برای تعریف رجیستری که همواره مقدار یک داشته باشد از دستور زیر استفاده می‌کنیم

reg p;

initial

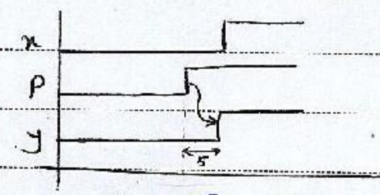
p=1;



PaPCO

۱.۴ ایجاد تأخیر: برای نشان دادن تأخیر از زمان تغییر تا زمان اجرا از علامت مربوط قبل از gate استفاده می‌کنیم. تا خروجی با تأخیر تولید بشود.

```
#5 or MO(y, x, p)
```

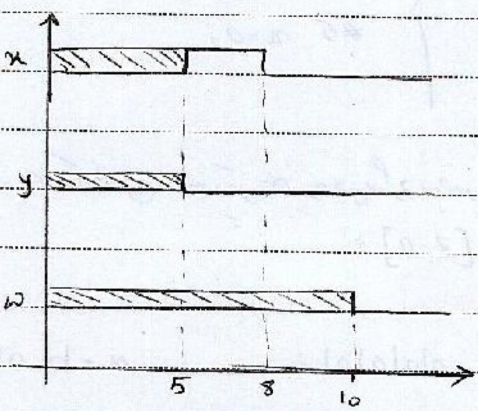


همچنان از این شیوه‌ی تأخیر برای ایجاد تأخیر در سیم نیز می‌توان استفاده کرد:

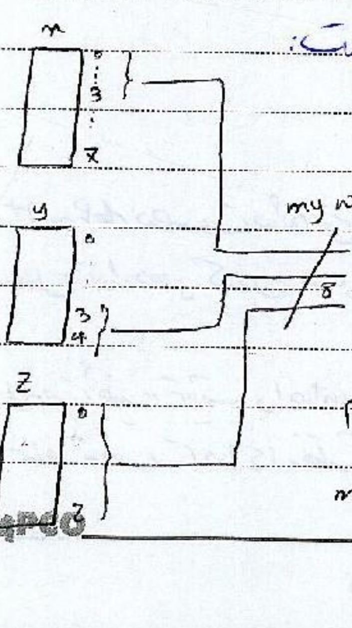
```
#5 wire x;
```

مثال:

```
initial
  begin
    #5 x=1;
       y=0;
    #3 x=0;
    #2 w=0;
  end
```



مثال: ایجاد یک باس 8 بیتی از رجیسترهای مختلف به صورت زیر است:




```
wire myw[7:0];
assign myw = {x[1:3], y[3:4], z};
```

زمانی که می‌خواهیم به wire مقداری دهیم

یعنی با تغییر مقدار هر یک از رجیسترها my w نیز عوض می‌شود.

Subject:
Year. Month. Date. ( )

نکته: تأخیر در زمانی که از صفر به یک می‌رود و برعکس با یکدیگر متفاوت است برای نشان دادن این امر از دستور زیر استفاده می‌کنیم.

```
# (5,4) and MA(x,y,z);
```

تأخیر از صفر به یک

تأخیر از یک به صفر

نکته: می‌توان تأخیر به صورت مجزا نیز نوشته شود و سپس عملیات را اجرا کنیم.

```
begin
  x=1;
  #5;          →   x=1;
  x=0;             #5 x=0;
end
```

نکته: برای مشخص کردن type متغیری که داخل رجیستر ذخیره می‌شود به صورت زیر عمل می‌کنیم.

```
reg x[7:0];
initial
  x=b'01010101;   →   x=b'0101_0101;
  x=O'54;
```

می‌توان الگوی های مختلف جهت خوش خط شدن استفاده کرد و تأثیری در برنامه ندارد.

default در صورت مشخص نشدن نوع آن decimal است. بهتر است تأخیرهای ایجاد شده توان مقداردهی کردن در test bench باشد.

```
reg x,y,z;
initial
  begin
    #3 x=1;
    #2 y=0;
    #1 x=0;
    #4 y=1;
  end
```

sequential

ایجاد تأخیر به ترتیب را sequential مطابق سمت راست است که بهتر است برنامه به صورت موازی رفتار کند.

اجرای موازی دو دستور fork join

```
fork
  #3 x=1;
  #2 y=0;
  ⋮
join
```

یعنی در لحظه t همه دستورات بین fork و join تا پایان اجرا شدن باهم اند ولی در اولی دستورات به ترتیب اجرا می‌شود.

نکته: می‌توان از ترکیب fork-join و begin-end استفاده کرد.

```
fork
  #3  x=1;
  #2  y=6;
  begin
  #1  x=~x;   } → به صورت sequen. اجرا می‌شود
  end
  #4  y=1;
join
```

5) bufif :

برای تعریف buffer از این دستور استفاده می‌کنیم:

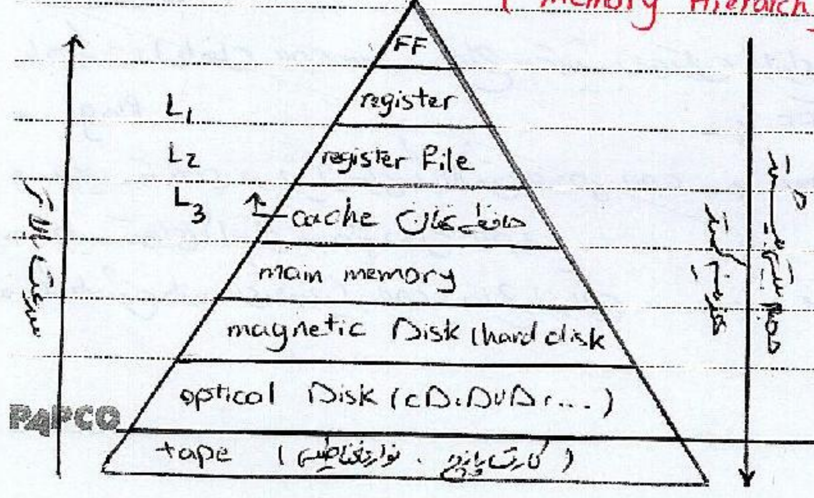
a → (buffer, c) → b ⟶ bufif

?? controle

bufif (b, c, a)

نکته: در سخت افزار نمی‌توان تأخیرها را پیاده‌سازی کرد و تنها برای شبیه سازی و نزدیک شدن به واقعیت است.

سلسله مراتب حافظه: (memory Hierarchy)



حافظه	delay
C_1	$d_1 \sim 5ns$
C_2	$d_2 \sim 5ns$
C_3	$d_3 \sim 5ns$
C_4	$d_4 \sim 10ns$
C_5	$d_5 \sim ms$
C_6	$d_6 \sim ms$
C_7	$d_7 \sim s$
C_8	$d_8 \sim s$

Subject:
Year. Month. Date. ( )

* تمرین: مفاهیم زیر چه نسبت و چه تفاوتی با یکدیگر دارد؟

cycle time ، Access time

۱ـ FF : یک بیتی را در خود نگه می دارد ولی نمی توان برای حافظه ها با ظرفیت بالا از FF استفاده کرد

۲ـ register file : تعدادی از رجیستر ها که داخل هم پردازنده ها وجود دارد و به صورت آرایه ای است ولی نسبت به رجیستر سرعت کمتری دارد

۳ـ main memory : تعداد بیت های بیشتری را نسبت به قبلی ها نگه می دارد ولی سرعت دسترسی کم است

در جدول قبل هر چه به سمت سطوح بالا پیش می رویم سرعت بیشتر می شود. یعنی که نیاز به حافظه ای داریم که اطلاعات را ذخیره کند و به طور موقت در خود نگه دارد، همچنین سرعت بالایی هم داشته باشد و هزینه کمتری مصرف کنیم از حافظه های سلسله مراتبی استفاده می کنیم یعنی x% از هر حافظه را مصرف می کنیم.

$c_1 < c_2 < c_3 < \dots < c_9$ → از بالا به پایین ظرفیت حافظه بیشتر

$d_1 < d_2 < d_3 < \dots < d_9$ → از بالا به پایین تأخیر بیشتر می شود

در حافظه دو ویژگی مهم وجود دارد:

۱ـ Access time : از لحظه ی درخواست تا لحظه ای که حافظه داده را آماده می کند

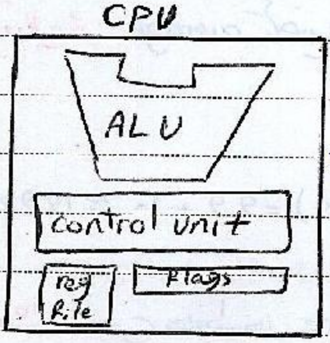
2ـ cycle time : زمان انجام یک فرایند (از زمانی که پردازنده به حافظه دستور می دهد تا زمانی که دوباره بتواند دسترسی بعدی را شروع کند)

دو ویژگی دیگر در حافظه read time و write time است که معمولاً سرعت read از write بیشتر است.

با حرکت از داخل cpu به سمت خارج به ترتیب می توان از انواع مختلف حافظه استفاده کرد:

* Flag ← FF
* رجیستر های مورد نیاز برای نگه داشتن خروجی cpu ← register
* رجیسترهای آرایه ای و منظم درون cpu ← register file
* نگه داشتن داده های مورد نیاز cpu در خارج از آن ← cache ، main memory

Subject:
Year. Month. Date. ( )



برای استفادهٔ cpu از hard disk روش‌های مختلفی وجود دارد:

۱- دسترسی مستقیم به هارد دیسک بدون استفاده از حافظه‌های level بالاتر

این روش سرعت بسیار پایینی دارد و کند عمل می‌کند.

۲- استفاده از سطوح بالا به پایین سلسله مراتبی درون پردازنده برای داشتن سرعت بالا

ما به دنبال حافظه‌هایی هستیم که بتوانند hit بالایی داشته باشند:

- hit (موفقیت) ← حافظه بتواند به درخواست داده شده پاسخ دهد
- miss (شکست) ← حافظه نمی‌تواند به درخواست پاسخ دهد

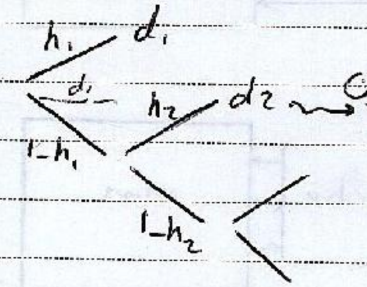
موفقیت + شکست = کل درخواست‌ها

نکته: hit و اندازهٔ حافظه با هم رابطهٔ مستقیم باید داشته باشند یعنی هر چه فضا بیشتر شود hit بالاتری خواهیم داشت. با وجود اینکه در level های بالا miss بیشتری دارند ولی در ساختار کنونی cpu از سلسله مراتبی استفاده می‌کنیم.

%0 = miss آخرین سطح   %100 = hit آخرین سطح

هر چه به بالای عمودی برویم رابطه عکس می‌شود

تأخیر دسترسی به data در پردازنده



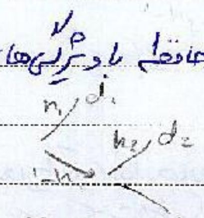
اختلاف d1 و d2 بسیار زیاد است پس می‌توان از d1 صرف نظر کرد.

تأخیر دسترسی به data $= h_1d_1 + (1-h_1)\left[h_2d_2 + (1-h_2)(h_3d_3 + (1-h_3)(\dots))\right]$

Subject:
Year. Month. Date. ( )

مثال: average تأخیر دسترسی دو حافظه با ویژگی‌های زیر وجود دارد:

delay	hit
10 ns	99% h_1
1000 ns	100% h_2



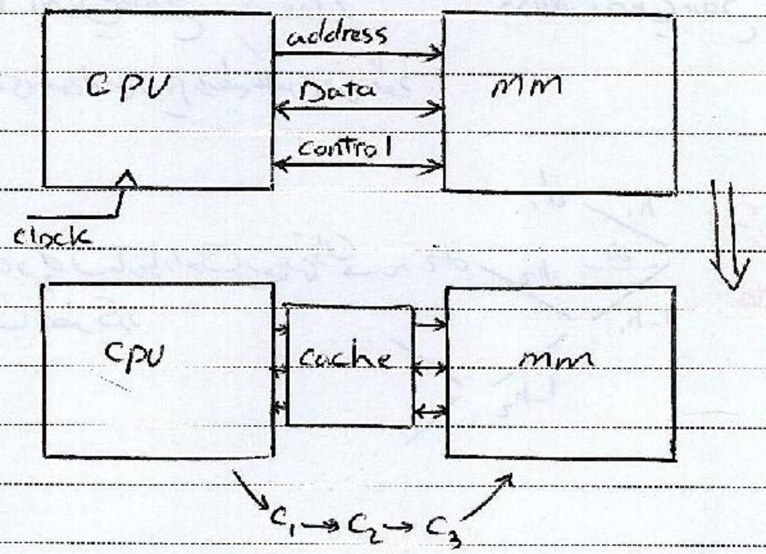
$$= 0.99\,(10\,ns) + 0.01\,(1000\,ns) = 9.9 + 10 \simeq 19.9\,ns$$

$$h_1 d_1 + (1-h_1)(h_2 d_2) = 0.99 \times 10 + 0.01 \times 1 \times 1000 \ldots$$

نکته: از بین حافظه‌ها tape ها از نوع sequential هستند و magnetic و optical از نوع Random Access در دسترسی به اطلاعات می‌باشند.

Cache:

بسیاری از درخواست‌هایی که از cpu به mm می‌آید قابل پیش‌بینی هستند. بدین گونه که با قرار دادن یک observer بین cpu و mm درخواست‌های cpu را کنترل کردند و مشاهده شد که بعضی از درخواست‌ها ترتیبی یا تکراری هستند. در نتیجه در بین آن دو یک حافظه به نام cache قرار دادند تا درخواست‌های احتمالی را در خود نگه دارد.



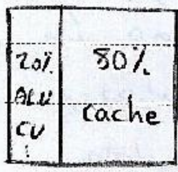
می‌توان از چندین cache در level های مختلف استفاده کرد.

PAPCO

برای بالا بردن سرعت CPU از مکانیزم های زیر استفاده کردند:

۱ـ clock پردازنده را کاهش دادند ولی از یک جا به بعد نمی‌توان clock را کم کرد زیرا آنقدر کم بود که قبل از انجام عملیات clock عوض می‌شد

۲ـ بالا بردن حافظهٔ cache



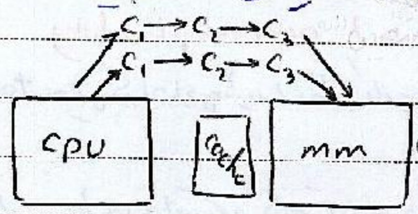
۳ـ استفاده از چندین cache که در حال حاضر درون CPU نیز cache وجود دارد:

on-chip cache ← درون CPU سرعت بالاتر و دسترسی بیشتر

off-chip cache ← خارج از CPU

نکته: cache level بر اساس نزدیکی به CPU مشخص می‌شود که هرچه به level بالاتر رویم حجم بیشتر و هزینه نامرغوب‌تر می‌شود (منظور از بالا یعنی سطوح پایین‌تر سلسله است)

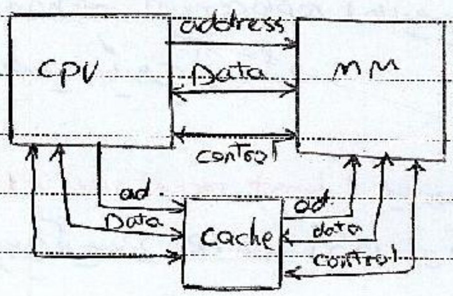
۴ـ instruction و data را در بین درخواست های CPU جدا کردیم زیرا تداخل بین هر دو بیشتر بود



شکستن داده از دستور به صورت ساختار command ها مشخص می‌شود مثلاً بعد از دستور Add حتماً دو داده وجود دارد پس برای آن دستورات داده ها تقسیم می‌شوند.

۵ـ اضافه کردن تعداد CPU های کامپیوتر.

moore's Law: تعداد ترانزیستورها در واحد سطح هر 18 ماه، ۲ برابر می‌شود یعنی ابعاد ترانزیستور کم می‌شود.



$$\text{hit ratio} = \frac{\text{تعداد hit}}{\text{تعداد کل درخواست‌ها}} = \frac{\#hit}{\#hit + \#miss}$$

$>90\%$ $= \frac{99}{96}$

Subject:
Year. Month. Date. ( )

درخواست هایی که از طرف cpu می آید دارای Locality یا همجواری هستند که شامل سه دسته زیر می شوند:

۱- spatial Locality (همجواری مکانی):
دستورهایی که پی در پی به ترتیب اجرا می شوند در واقع اصل برنامه ها sequential هستند. فقط ممکن است یک jump رسیده و نیاز باشد درخواست قبل اجرا شود یا part که چندین بار اجرا می شود.
پس درخواست های cpu از نظر مکانی به هم نزدیک هستند.
data در قسمت خاصی از حافظه وجود دارد.

۲- temporal locality (همجواری زمانی):
درخواست های cpu از نظر زمانی به هم وابسته هستند یعنی هر چند وقت یک بار به آدرس خاصی درخواست می شود.
نکته: locality = همجواری (درخواست های cpu random نیست و نظم خاصی دارد.)

۳- process locality (همجواری فرآیندی):
task هایی که اجرا می شوند از نظر مکانی به هم نزدیک هستند ؟؟

چگونه از cache استفاده کنیم تا با وجود خصوصیات بالا hit ratio بالاتری داشته باشیم؟

۱- placement (جایدهی): داده جدید را چگونه درون حافظه قرار دهیم که این الگوریتم تا زمانی که cache خالی است بسیار مهم است.
۲- replacement mechanism (جایگزینی): پس از پر شدن cache کدام داده را انتخاب کنیم که داده جدید جایگزین آن شود.

LRU: (least recent use) یکی از روش ها برای انتخاب داده ای که از cache حذف شود تا داده جدید جایگزین آن شود LRU است که داده ای که طول عمر زیادی دارد و از آن استفاده نشده است انتخاب می شود.

Subject.
Year. Month. Date. ( )

روش های placement :

درخواست هایی که از طرف cpu می‌آید که مربوط به آدرس های حافظه است را چطور این آدرس ها را در cache ذخیره کنیم که مسئله ترجمه ← address mapping

1- hash table : از آنجایی که پیاده‌سازی hash func در سخت افزار امکان پذیر نیست از این روش استفاده نمی‌شود.

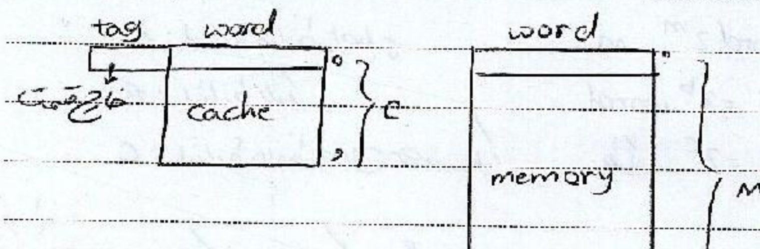
2- Direct mapped (نگاشت مستقیم) : باقیمانده آدرس های حافظه را بر سایز cache حساب کرده در نتیجه هر خانه از حافظه در یکی از خانه های cache نگاشت می‌شود. این الگوریتم کاملا بدیهی است.

آدرس در حافظه نهان = آدرس در حافظه اصلی Mod C

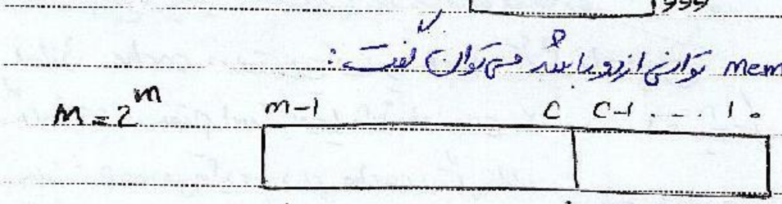
↳ اندازه cache

3- Associative (انجمنی) : قرار گرفتن داده ها در cache بدون هیچ اولویتی انجام می‌گیرد و هر جایی که خالی بود آدرس در آنجا قرار می‌گیرد. در این روش برای پیدا کردن آدرس مورد نظر باید search کنیم.

نکته: در روش نگاشت مستقیم برای فهمیدن اینکه آدرسی که در خانه مورد نظر از cache وجود دارد مربوط به داده مربوط است یا نه خارج قسمت تقسیم آدرس بر اندازه حافظه cache را در tag نگه می‌داریم.

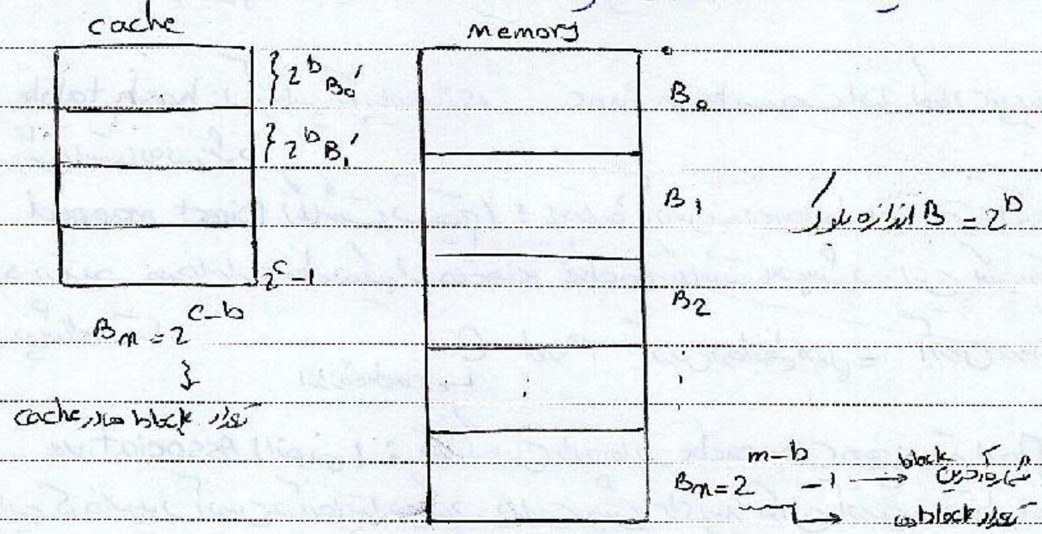


در این حالت اگر تعداد خانه های memory توانی از دو باشند می‌توان گفت:



Subject:
Year. Month. Date. ( )

۱. در روش ذکر شده برای spatial locality در ویژگی direct mapped رعایت نشده است و برای دستورهای متوالی — سلسله memory می رود پس بهتر است به جای درخواست 1 word از cpu، cache یک block را در خود ذخیره کند.



قالب آدرس حافظه اصلی (مثلاً یک آدرس 32 بیتی در حافظه اصلی)

m-1 …	…	b b-1 … 1 0
	index	offset
برچسب	آدرس بلوک در حافظه اصلی	آدرس درون بلوک

آدرس درون = $\times$ / آدرس بلوک

x mode $C = y$

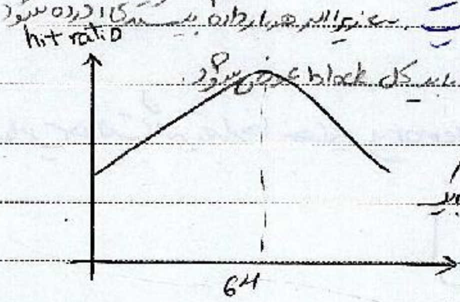
آدرس بر حسب بلوک در حافظه اصلی — آدرس بر حسب بلوک در cache

M : اندازه حافظه اصلی $= 2^m$ word

B : اندازه بلوک $= 2^b$ word

C : اندازه حافظه کش بر حسب بلوک $= 2^c$ بلوک

hit ratio $\simeq 92\%$



نکته ۵: اندازه بلوک هر چه بزرگتر باشد بهتر است ولی با محدودیت. به طوری که اگر هر اندازه بیشتر از آن در نظر گرفته شود امکان پاسخگویی بیشتر است ولی با بزرگ شدن اندازه block ها دسترسی به آن عدم وجود به کل block ها منجر می شود.

اندازه cache در بیشترین بار block تاثیر دارد چون

مناسب نیستم است زیرا مثلاً خانه های ۰، ۲، ۴، ۶ همه به یک خانه map می شود در بار cache مشکل دارد.

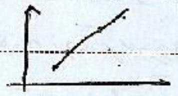
Subject: Simple-scalar www.spec.org

Year. Month. Date. ( )

۱۵ فروردین

* تمرین: شبیه‌سازی حافظه نهان Direct mapped با استفاده از ابزار Sim_watch عددی

حداقل ۳ نمودار کاری گزارش میزان توان مصرفی، مساحت، hit ratio با توجه به تغییر هر یک از پارامترهای زیر



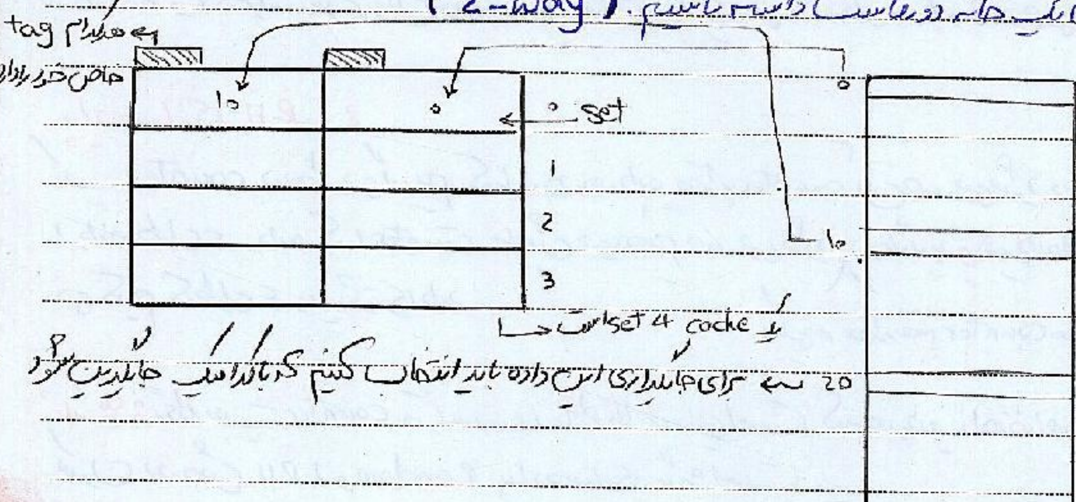
الف) اندازه حافظه نهان

ب) اندازه بلاک در حافظه نهان

نکته 8: در cache می‌توان داده را همراه آدرس نگه داشت. با ترتیب مکانیزم‌های مطرح شده خانه‌های حافظه اصلی را در cache نگاشت می‌کنیم و سپس در خانه محاسبه شده، داده و آدرس آن را نگه می‌داریم. علت جدا کردن دو بایت از سمت راست در آدرس برای این است که داده‌ها هم جواری مکانی داشته باشند. مثلاً اگر از سمت چپ آدرس دو بایت جدا شود خانه‌ها از بلوک‌های مختلف در ستون انتخاب می‌شوند در cache قرار می‌گیرد که هیچ ارتباطی با هم ندارند.

مشکلات direct map: خانه‌های زیادی از حافظه اصلی باید یک جا هم‌پوشانی شوند و هستند و دسترسی به حافظه RAM بیشتر می‌شود در حالی که بعضی از قسمت‌های حافظه نهان همچنان خالی مانده است.

12 برای برطرف کردن مشکل بالا اندازه حافظه نهان را نصف کرده و در کنار ردیف قبلی قرار می‌دهیم تا بتوانیم برای یک خانه دو نگاشت داشته باشیم. (2-way)



20 ← برای جایگذاری این داده باید انتخاب کنیم که با کدام یک جایگزین شود

(اندازه cache از نظر عمودی نصف می‌شود و یک ستون به آن اضافه می‌شود.)

سیاست های جایگزینی (Replacement policy) :

1) FIFO : داده ای که اول آمده زودتر بیرون می رود.

2) LIFO : داده ای که دیرتر آمده زودتر بیرون می رود

3) Random : داده ای را به صورت رندم انتخاب کرده و جایگزین می نماییم

4) Distance : آنکه فاصله ی بیشتری دارد خارج شود.

5) LRU (Least recent used) : تعداد استفاده های اخیر از داده را محاسبه کرده داده ای که کمتر استفاده شده را بیرون می اندازیم.

6) MRU (most recent used) : داده ای که بیشترین استفاده از آن شده دیگر دیرتر نیاز می شود پس آن جایگزین می کنند.

7) LFU (least Frequently used) : داده ای که کمترین فرکانس استفاده را دارد. که از زمان شروع سیستم تعداد استفاده ها را در نظر می گیریم.

8) MFU : داده ای که از شروع سیستم تا حالا بیشترین استفاده از آن شده دیگر نیاز نخواهیم داشت.

نکته : روش 7 دقیق تر است و منطقی تر می باشد ولی در LFU باید برای هر داده یک بافر بزرگ جمع آوری کنیم که قابل پیاده سازی نیست. روش 5 بیشتر می دانیم اخیراً را چه مدت زمانی در نظر بگیریم
5 ، 6 و 7 براساس زمان و در بازه ی زمانی تصمیم گیری می کنند ولی در 1 و 2 توجهی به زمان نمی شود.

پیاده سازی LRU :

یک counter در نظر می گیریم که با ورود هر داده مقدار counter آن صفر می شود و برای هر بار استفاده از داده طول عمر داده که با counter نمایش می دهیم صفر می شود. در جایگزینی نیز داده ای را انتخاب می کنیم که طول عمر بیشتری دارد.
ولی به مقدار بقیه counter ها اضافه می کنیم.

نکته : تعداد بیت counter متناسب با تعداد خانه هایی است که به هر ردیف اختصاص داده شده است.
اکنون در روش LRU و Random پیاده سازی شده است.

Subject:
Year. Month. Date. ( )

با تقسیم cache و افزایش تعداد خانه‌ها در هر سطر طول cache کاهش می‌یابد. بسته k-way

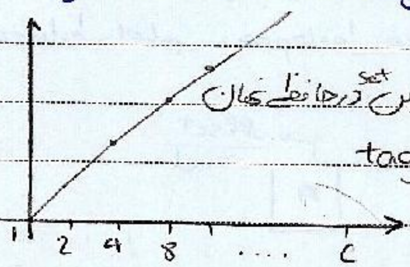
k-way ↓ توانی از ۲

یک c line و خانه max ⟶ C-way

(placement الزام)

3) k-way set Associate : (KWSA)

با اضافه کردن تعداد ستون‌ها در cache ، قابلیت پاسخ‌دهی آن بیشتر می‌شود تا جایی که cache به یک ردیف با C خانه تبدیل می‌شود.

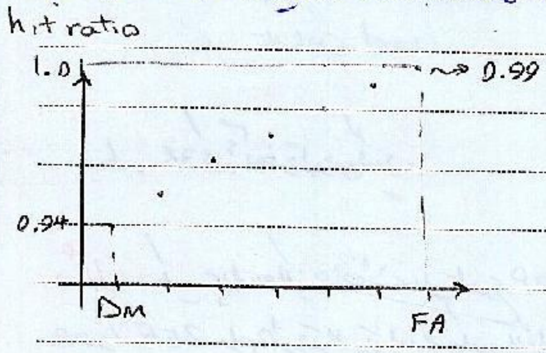


آدرس set در حافظه نهان = آدرس در حافظه اصلی Mod C/K

tag = آدرس در حافظه اصلی Div C/K

4) Fully Assocative (FA) :

در این حالت اندیس وجود ندارد و هر داده می‌تواند هر جای خالی که وجود دارد قرار بگیرد. و اگر خانه خالی وجود نداشت از روش‌های دیگر شده جایگزین می‌شود



k=2 hit ratio ≃ 95%

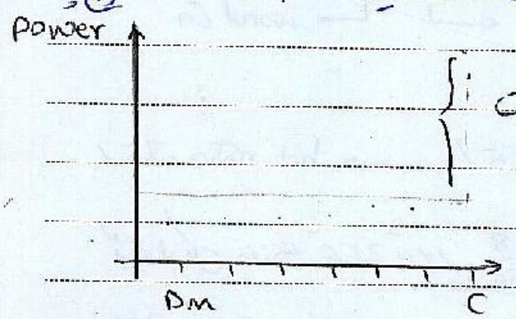
k=4 hit ratio ≃ 96%

⋮

مشکلات FA : ۱- برای پیدا کردن یک داده باید تمام tag ها را search کند تا داده مورد نظر را پیدا کند پس برای پیاده‌سازی نیاز به حافظه CAM داریم ولی توان بالایی دارد. بطوری که در هیچ پردازنده‌ای از FA استفاده نمی‌شود

مشکلات :
- power توان مصرفی
- Area مساحت اشغالی

اختلاف زیادی بین توان مصرفی C و DM وجود دارد.



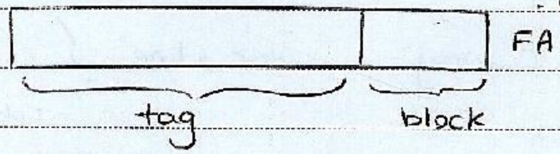
PAPCO

33

* تمرین 4 : قالب آدرس برای k-way set associative را بدست آورید.

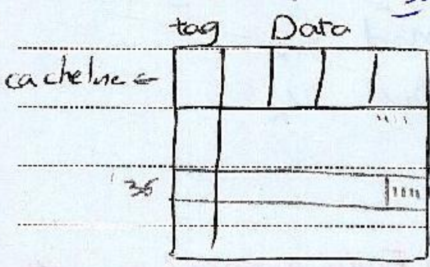
Subject.
Year. Month. Date. ( )

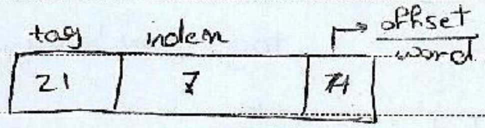
۲- اندازه tag در FA خیلی بزرگتر از DM است. زیرا اندازه ایندکس صفر است و داده هر جایی می‌تواند قرار بگیرد. (کل آدرس tag است).



مثال:

در صورتی که دسترسی به حافظه کش از طریق مشخصات زیر یافت شده است و مستقیم باشد مشخص کنید حجم حافظه اصلی، حجم حافظه کش و محل استقرار آدرس زیر 000B3A4F h



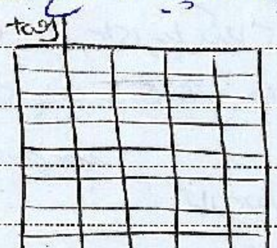


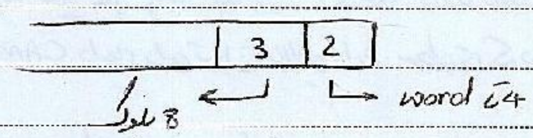
2^{32} ← حجم حافظه اصلی

$2^7 \times 2^4$ ← 2^{11} ← حجم حافظه کش بر اساس word

سطر 36 ام در آخر قرار می‌گیرد.

مثال: یک پردازنده یک حافظه کش از نوع direct map با 8 بلوک 4 کلمه‌ای است. برنامه آدرس‌های صفر تا 255 را به طور متوالی تولید می‌کند. اگر این کار تا بی‌نهایت ادامه پیدا کند نرخ برخورد را بدست آورید.





$\frac{8}{32} = \frac{1}{4} = 25\%$ ← hit ratio = 75%

32 بار درخواست که در هر بار همین تعداد اتفاق می‌افتد و صورت [illegible] نرخ ناموفق

اگر تعداد کل خانه‌ها 256 تا بود $\frac{64}{256}$ ولی این تعداد در یک بار است چون این روند تا بی‌نهایت تکرار می‌شود نرخ به سمت بی‌نهایت می‌رود

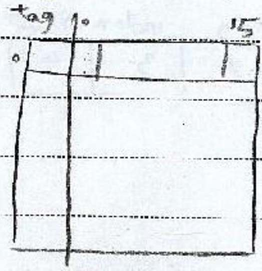
PAPCO

platform

Subject: دستورات

Year. Month. Date. ( )

مثال: حافظه‌ی نهان از نوع Direct map به گنجایش 256 بایت داریم که هر بلوک آن 16 بایتی است. پردازنده‌ای آدرس‌های 16 بیتی تولید می‌کند. ارجاع‌های زیر را از ابتدا تا انتهای لیست دو بار تکرار می‌شود. miss ratio را به دست آورید.



$$\frac{2^8}{2^4}=2^4$$

tag		
6	4	4

$$\frac{2}{10}=\frac{1}{5}=20\%$$

×FF00 hex
✓FF01
✓FF02
✓FF03
✓FF04
×5F00
5F01
5F02
5F03
5F04

مثال: در یک کامپیوتر فضای آدرس‌دهی حافظه 2^{28} می‌باشد. یک حافظه‌ی نهان با 4 بلوک در هر دسته (4-way associative) 1024 بلوک و هر بلوک 4 بایت است. محتویات آدرس زیر در کدام بلوک قرار می‌گیرد.

39 | 3

888809F h

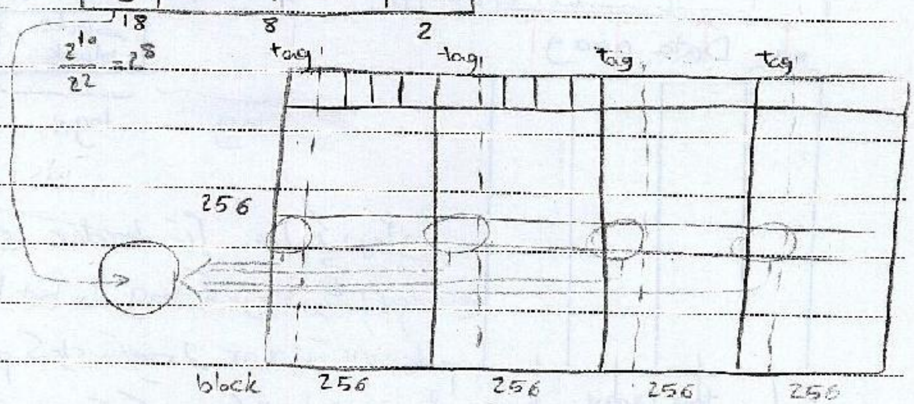
جا 39 در خانه 3

در هر block می‌تواند قرار بگیرد

28

tag	index/set	offset
18	8	2

$$\frac{2^{10}}{2^2}=2^8$$



به عنوان آدرس index آن را از بین 256 تا پیدا کنید سپس وجه tag خود را با 4 tag حافظه مقایسه می‌کند

تعداد سطرها مشخص است ولی نمی‌دانیم در کدام حافظه قرار داده بیاندازیم

PAPCO

* تمرین: شبیه سازی حافظه‌های { set associative ? / Fully associative ?? } با استفاده از شبیه ساز simwatch

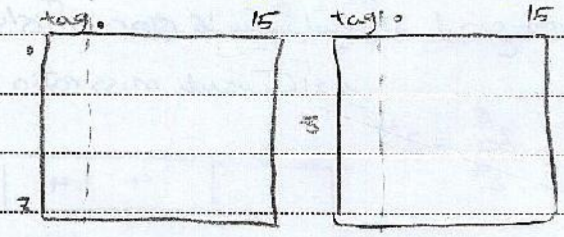
Subject:

Year. Month. Date.

گروهی حداقل ۳ برنامه کاربردی که دارای کران محدودی ، کارایی ، مساحت با توجه به تغییرات هر یک از روش‌ها کار کنید

مثال: دو مثال قبل را برای حافظه انجمنی 2-way حل کنید.

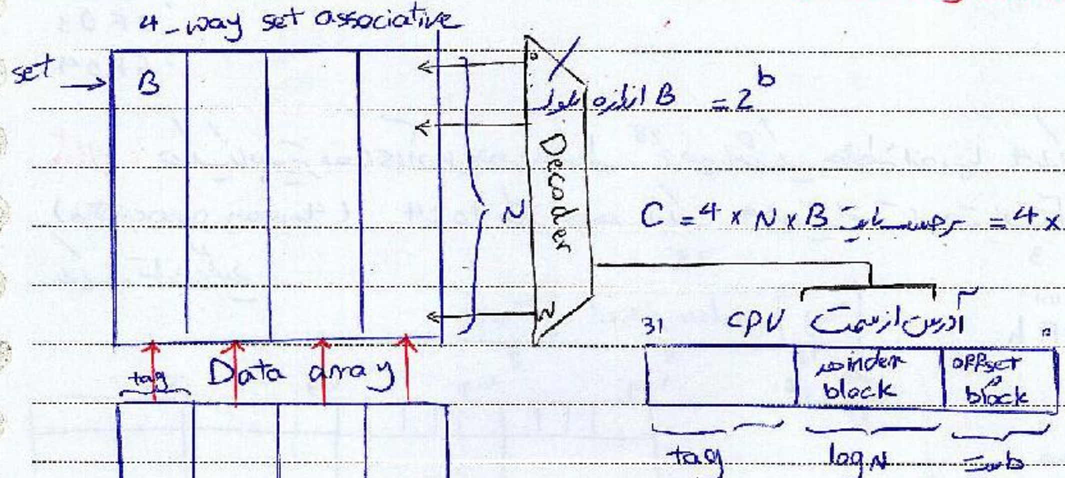
FF00 ×
FF01 ✓
FF02 ✓
FF03 ✓
FF04 ✓
5F00 ×
5F01 ✓
5F02 ✓
5F03 ✓
5F04 ✓



tag	index	offset
9	3	4

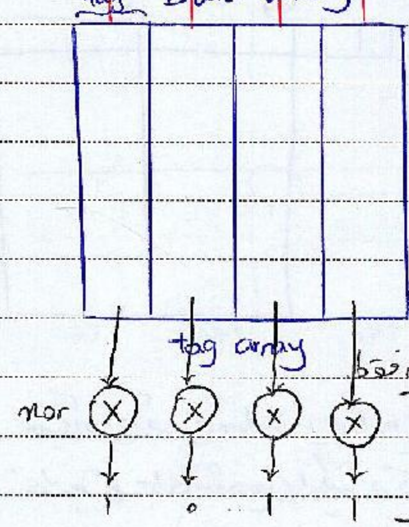
$\frac{2}{100} = 2\%$

سازمان k-way associative :



B اندازه بلوک $= 2^b$

$C = 4 \times N \times B$ = ترجمه سایز   بلوک $= 4 \times N$



با فعال شدن یکی از پایه‌های decoder آن line فعال می‌شود

برای هر bit line ها tag آدرس مربوط را با خانه‌های tag array مقایسه می‌کنیم که برای مقایسه از nor استفاده می‌کنیم.

دستور مقایسه کردن این است که یا هیچ کدام match نمی‌شود یا فقط با یکی از آنها برابر است.

→ اگر دو tag باهم برابر باشند خروجی صفر و اگر برابر نباشد خروجی nor یک می‌شود.

به یک مالتی‌پلکسر برای خارج شدن داده معادل با tag مربوط در مقایسه offset در بلوک نیاز داریم

36

الف) اندازه حافظه نهان (K=4)　　ج) با افزایش K در set associative

ب) اندازه بلوک (K=4)

Subject:
Year.　Month.　Date.　( )

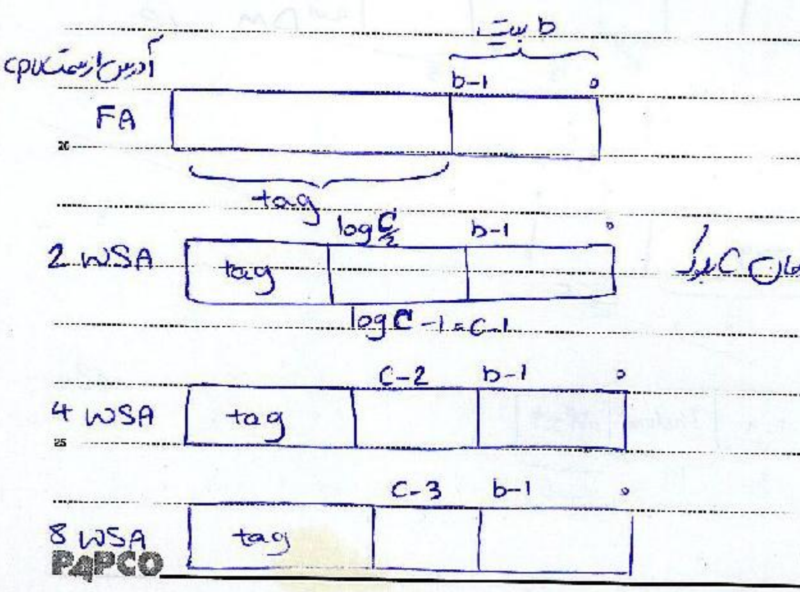
نکته: طراحی اولیه این نوع از حافظه‌ها این گونه بود که همه خانه‌های tag array برای خود یک مقایسه کننده داشتند و خروجی ایشان هم دارند ولی هر line آنها زمانی که روشن باشد فعال می‌شود سپس نیازی نیست که همه خانه‌ها مقایسه کننده داشته باشند برای همین امروزه از یک سری مقایسه کننده برای همه خانه‌ها استفاده می‌کنیم.

در Fully associative با وجود اینکه نیازی به decoder نداریم ولی تعداد مقایسه کننده‌ها زیاد می‌شود و مدار کنترلی برای بدست آوردن داده متناظر با tag مربوطه پیچیده می‌شود. بیشترین tag در FA وجود دارد و قالب آدرس آن تنها دو قسمت دارد.

	کارایی	توان مصرفی	مساحت	Access time (تأخیر)
DM	−	−	−	−
KSA	+	+	+	+
FA	++	++	++	++

با وجود اینکه تأخیر در FA بیشتر است ولی در عوض اصلاً داده در حافظه نهان وجود دارد درحالی که با وجود کم بودن تأخیر در DM ولی در بسیاری از موارد داده در حافظه نهان وجود ندارد.

ساختار Fully associative:



اندازه حافظه نهان C بلوک $= 2^c$

Subject:
Year. Month. Date. ( )

DM

tag	c	b-1

→ کمترین اندازهٔ tag را دارد.

نکته 8 مشکل است بعد از روشن کردن کامپیوتر مقادیری از قبل در cache مانده باشد و در ابتدا که هنوز cache کامل پر نشده است ممکن است آدرس خارج شده از cpu به اشتباه با یک خانه یکی شود برای جلوگیری از این اشتباه از یک بیت valid استفاده می کنیم که تمام یک ها با این بیت استفاده می شود.

مشکل دوم این است که cache باید هر دفعه که می خواهد یک داده را خارج کند در صورت replacement باشد باید RAM را نیز update کند در صورت خراب شدن یکی از بیت ها در cache دیگر دسترسی به آدرس نداریم و مقدار اشتباهی در RAM قرار می گیرد. به این روش write_back می گوییم روش دیگر write_through است که بلافاصله بعد از آمدن دستوری داخل cache تغییرات را در RAM منتقل کند و آن را update کند.

مثال: فرض کنید حافظهٔ اصلی به اندازهٔ 16 MB ($2^{4} \times 2^{20} = 2^{24}$) و حافظهٔ نهان مجموعه انجمنی ۲ تایی با اندازه 512 KB می باشد چنانچه اندازهٔ بلوک 32B (2^5) باشد مطلوبست اندازهٔ فیلدهای مختلف در قالب آدرس.

الف) 2 WAS

ب) FA

ج) DM

(الف)

23

5	13	5

$$\frac{2^{9} \times 2^{10}}{2 \times 2^{5}} = 2^{13}$$

ب)

23

tag	5

$$\frac{2^{19}}{2^{5}} = 2^{14}$$

ج)

23

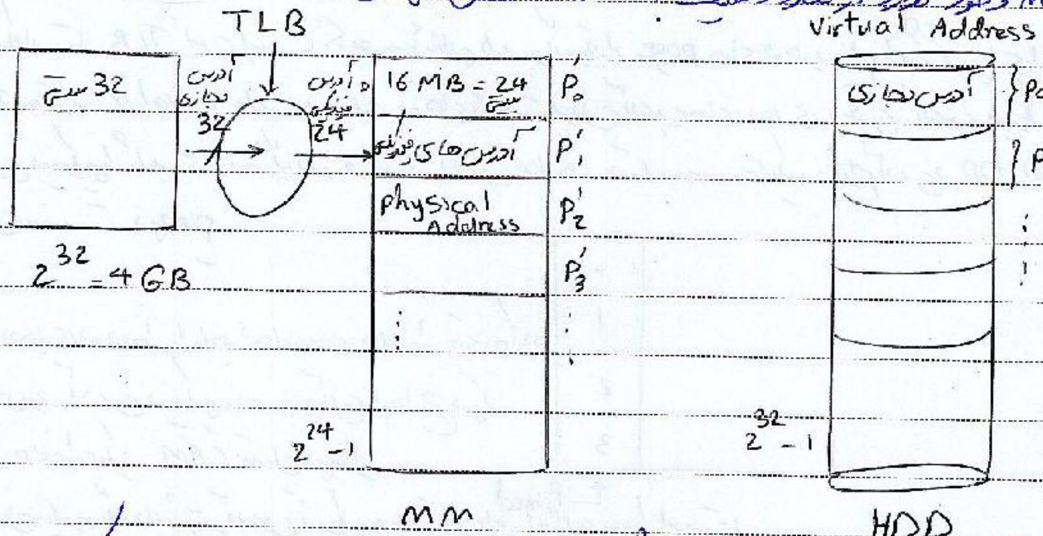
tag	Index	offset
5	14	5

Subject:
Year. Month. Date. ( )

TLB:

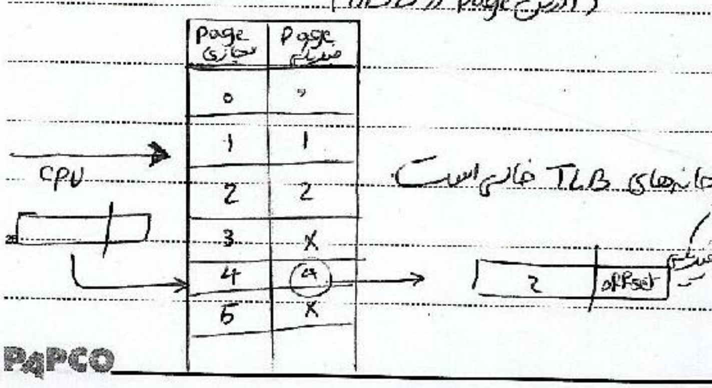
سازمانی در ارتباط با cache که در پردازنده‌ها وجود دارد و مخفف Translation Lookaside Buffer است.

CPU می‌تواند $2^{32} = 4GB$ را پردازش کند ولی در RAM به این اندازه حافظه وجود ندارد که در این حالت آدرس‌های پردازنده براساس حافظه اصلی‌ها است و اگر آدرس مورد نظر در MM وجود نداشت از هارد دیسک به RAM منتقل می‌شود



مسئله‌ای که به وجود می‌آید این است که چگونه یک آدرس 32 بیتی را به 24 بیتی تبدیل کنیم که در این جا از TLB استفاده کردند. کار TLB این است که آدرس مجازی را به آدرس فیزیکی تبدیل کنیم. در ضمن انتقال داده از HDD به MM در حد انتقال page است.

آدرس مجازی $\xrightarrow{\div \text{ اندازه page}}$ آدرس page مجازی (آدرس page در HDD) $\xrightarrow{TLB}$ آدرس page فیزیکی (آدرس page در MM) ⟶ آدرس فیزیکی



Page مجازی	Page فیزیکی
0	9
1	1
2	2
3	X
4	2
5	X

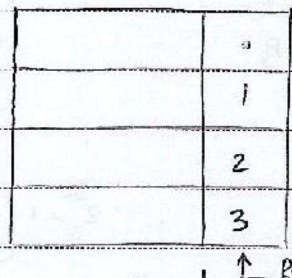
اگر page 4 در RAM جا داشته باشد بقیه خانه‌های TLB خالی است

Subject:
Year. Month. Date. ( )

برای تعیین مقادیر آدرس فیزیکی در هر خانه از TLB توسط memory managment انجام می‌شود و مقدارهای داخل جدول توسط آن update می‌شود.

نکته: cache بین TLB و RAM قرار دارد و لی حافظه‌های امروزی می‌توانند موازی با TLB نیز کار کنند.

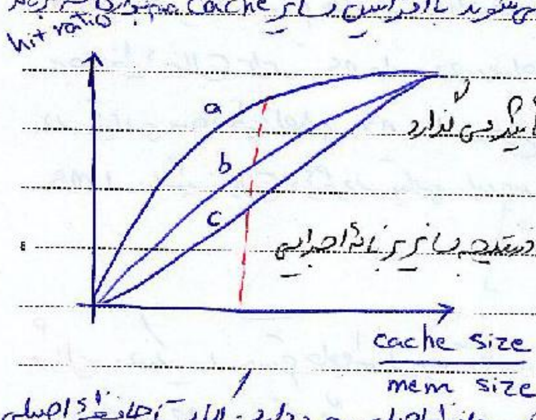
از مشکلات TLB این است که هزینهٔ بالایی دارد و اگر مقدار page های موجود کم باشند بسیاری از خانه‌های TLB مقدار unvalid دارد در نتیجه به جای ثابت بودن آدرس های مجازی و تغییر آدرس فیزیکی، آدرس فیزیکی را ثابت در نظر گرفته و مقدار آدرس مجازی را متناسب به اینکه کدام آدرس در HDD اشاره می‌کند به آن نسبت می‌دهیم.



آدرس مجازی مورد نظر را با همهٔ خانه‌های حافظه مقایسه کرده و اگر برابر بود آدرس فیزیکی مربوطه آن را خارج می‌کند در نتیجه حافظه یک CAM تبدیل می‌شود که دیگر نیازی به داشتن آدرس فیزیکی نیست و برابر با شماره سطر است.

شیوه موازی‌سازی cache و TLB به این صورت است که قسمت offset آدرس مجازی در همان ثابت باقی می‌ماند پس می‌توان تا زمان آماده شدن شماره page معادل در cache از طریق decoder حافظهٔ cache، چه اینکه بلوک و offset را تعیین کند و قسمت مقایسهٔ tag و تعیین شدن hit/miss بعد از تعیین شماره page از طریق TLB انجام می‌دهد.

مثال: سه برنامه a و b و c بر روی یک cache اعمال می‌شوند. با افزایش سایز cache منحنی‌های سه برنامه را با یکدیگر مقایسه کنید.



محلیت و هم‌جواری مکانی داده‌ها در میزان hit ratio تأثیر می‌گذارد.

پس هم‌جواری a خیلی بیشتر از b و c است.

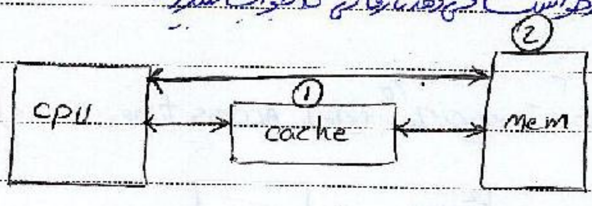
از طرفی چون a زودتر به مقدار ۱۰۰٪ رسیده است در نتیجه سایز برنامه اجرایی آن کوچکتر است.

مثال: در یک سیستم سلسله مراتبی یک cache و یک حافظه اصلی وجود دارد. کلمات حافظه اصلی 4 بایتی و حافظه از 64 بلوک و هر بلوک 8 کلمه است. همچنین cache هم 8 بلوک است. ساختار cache 2-way دارد. تعداد بیت‌های tag، set و word چیست؟

4	2	3

حافظه‌ها word می‌خوانند. کلمه‌ها هم پشت سر هم‌اند چون با word آدرس‌دهی می‌شود.

Access time: از زمانی که cpu درخواست می‌دهد تا زمانی که جواب بگیرد.



$$t_a = h_1 * t_1 + h_2(1-h_1)t_2$$

معمولاً hit ratio سطح آخر یک می‌شود و ۱۰۰٪ است یعنی $h_2=1$ است.

زمانی که cache و mem با هم درخواست را بگیرند access time کمتر می‌شود زیرا بعضی از درخواست‌ها بدون اینکه نیازی به mem باشد cache پاسخگو است که access time آن کمتر است.
ممکن است در بعضی از کتاب‌ها داشته باشیم:

$$t_a = t_1 + (1-h_1)t_2$$

که این مفروض بر این است که در ابتدا cache خالی است و در نتیجه همه زمان‌ها را cache می‌گذرانند.

PAPCO

Subject:

Year. Month. Date. ( )

مثال: یک سیستم حافظه شامل حافظه نهان، حافظه اصلی، حافظه ثانویه دارد. اگر زمان دسترسی به حافظه نهان برابر 10ns و درصد مراجعه و پیدا کردن اطلاعات مورد نظر 98% باشد و میانگین زمان دسترسی به حافظه اصلی 100ns و نرخ برخورد به آن 90% باشد و زمان دسترسی به حافظه ثانویه 1ms باشد زمان موثر دسترسی را بدست آورید.

$$0.98 \times 10 + 0.02 \times \left[0.9 \times 100 + 0.1 \times 10^6 \right]$$

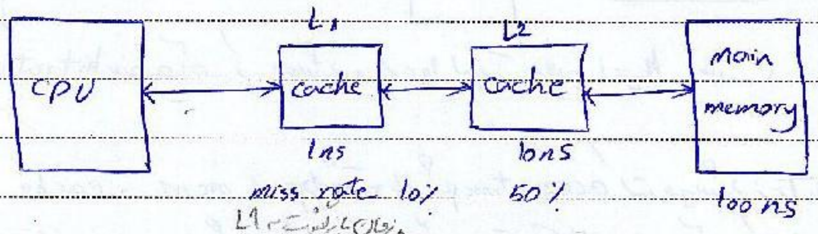
مثال: در یک سیستم حافظه سرعت دسترسی به cache برابر 100ns و سرعت دسترسی به حافظه اصلی برابر 1200ns می‌باشد اگر بخواهیم سرعت موثر دسترسی به حافظه بیشتر از 20% دسترسی به حافظه نباشد hit ratio برای cache باید چقدر باشد.

نکته: زمان موثر دسترسی همیشه بیشتر از cache و کمتر از زمان دسترسی به mem است.

$$t_a < \frac{20}{100} t_m \Rightarrow 100 + 1200(1-h_1) = 1300 - 1200h_1 < 0.2 \times 1200 = 240$$

$$1200h_1 < 1060 \Rightarrow h_1 < \frac{1060}{1200} = 88\%$$

مثال: Access time را برای شکل زیر بدست آورید.



زمان برداشت L1

$$0.90 \times 1 + 0.1 \left[0.5 (10 + 1) + 0.5 (100 + 10 + 1) \right] = 7$$

زمان برداشت L1, L2

در این حالت به خاطر عدم ارتباط مستقیم CPU و mm برای برگرداندن داده در CPU باید همین مسیر را برگردیم چون حافظه‌ها سلسله مراتبی هستند

PaPCO

Subject:
Year. Month. Date. ( )

مثال: برنامه‌ای روی کامپیوتر A اجرا می‌شود (در زمان 10 ثانیه) clock-rate کامپیوتر A 400 MHz است. همین برنامه در کامپیوتر B، 6 ثانیه طول می‌کشد. clock-rate کامپیوتر B را بدست آورید.

تعداد کلاک مورد نیاز برای B = 1.2 × A تعداد کلاک روی کامپیوتر → $I_C \times CPI_B = 1.2\, I_C \times CPI_A$

$B = 1.2A$

$$\text{Execution time} = \frac{I_C \times CPI}{f}$$

(I_C: تعداد دستورات — CPI: تعداد کلاک برای هر دستور)

$$10 = \frac{A}{400 \times 10^6} \Rightarrow A = 4 \times 10^9$$

$$6 = \frac{4 \times 10^9 \times 1.2}{f} \Rightarrow f = 8 \times 10^8 = 800 \times 10^6 = 800\ MHz$$

مثال: دستورالعمل‌های زیر در یک ماشین در سه کلاس موجود دارند. برنامه‌نویس برنامه را در دو صورت مختلف کامپایل می‌کند و دو کد مختلف به وجود می‌آید. CPI برای هر یک از این برنامه‌ها چقدر است

program	A	B	C	
1	2	1	2	= 5
2	4	1	1	= 6

کلاس	CPI
A	1
B	2
C	3

$$\sum \frac{CPI_i \times IC_i}{I_C} = \frac{1}{5}(1\times 2 + 2\times 1 + 3\times 2) = 2 \qquad \text{program 1}$$

$$\frac{1}{6}(4 + 2 + 3) = \frac{9}{6} = \frac{3}{2} \qquad \text{program 2}$$

از آن جایی که هر دو برنامه روی یک کامپیوتر اجرا می‌شوند فرکانس برای هر دو ثابت است پس می‌توان گفت برنامه دوم سریع‌تر است.