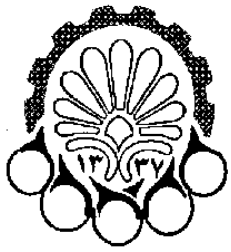


دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

جزوه درس

معماری کامپیوتر

Computer Organization & Design

نسخه 1.5

دکتر زرندی

ترم اول سال 1389

فهرست

فصل اول

مروری بر مدار منطقی

و حافظه‌های رایانه

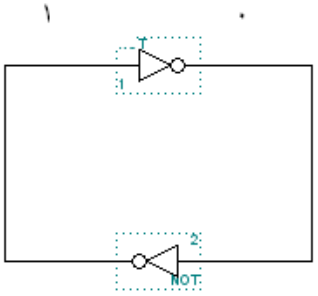
در این بخش در ابتدا یادآوری مختصـری از درس مدار منطقی می‌شود سپس با زبان verilog آشنا می‌شویم که برای استفاده از آن برنامه ModelSim پیشنهاد می‌شود که در ضمیمه 1 به طور جامعی مورد بررسی قرار گرفته است.

سـپس با حافظه‌های رایانه آشنا می‌شویم. در ابتدا حافظه‌ها را بر اساس سرعت و هزینه در سلسله مراتب حافظه مورد بررسـی قرار می‌دهیم و سپس می‌کوشیم تا با کمترین هزینه بیشترین سرعت را داشته باشیم. همانطور که می‌دانید حافظه اصلی ارزان اما سرعت آن کم است پس برای اینکه سرعت را بالا ببریم مقداری حافظه پرسرعت را میان حافظه اصـلی و پردازشـگر قرار می‌دهیم(cashe) و می‌کوشیم با شیوه‌های مختلف این ارتباط را سریعتر کنیم

سپس فاکتورهای کارایی یک سیستم مورد بررسی قرار می‌گیرد.

Latch/Flip Flop

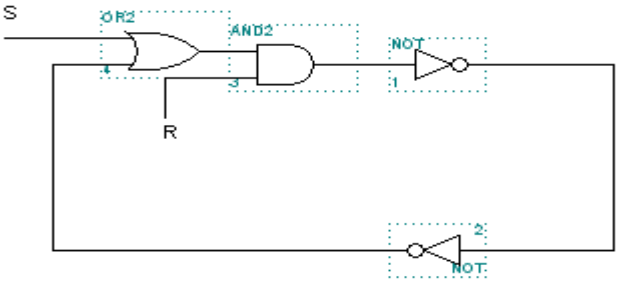
در مدار زیر می‌توانیم یک بیت اطلاعات ذخیره کنیم. اما هنگامی که مقدار داده شد دیگر نمی‌توان مقدار ذخیره شده را تغییر داد.



	R	S
نوشتن 0	0	0
حافظه ای	1	0
نوشتن 1	0	1
	1	1

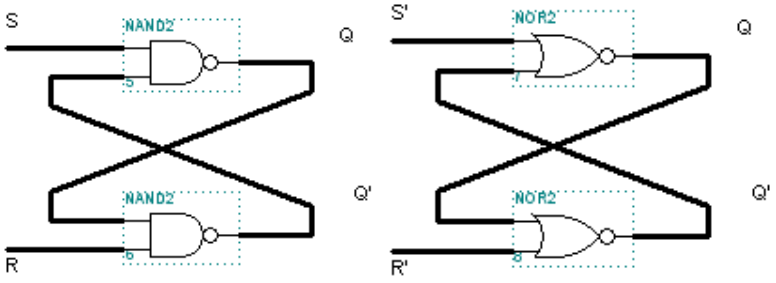
در مدار روبه رو می‌توان اطلاعات نیز ذخیره کرد، جدول صـــحت آن به شکل زیر است:

به این مدار Latch می‌گویند.



	R	S
تصادفی	0	0
نوشتن 1	1	0
نوشتن 0	0	1
حافظه ای	1	1

معمولاً رایج است که مدار Latch را با nand و nor می‌سازند.

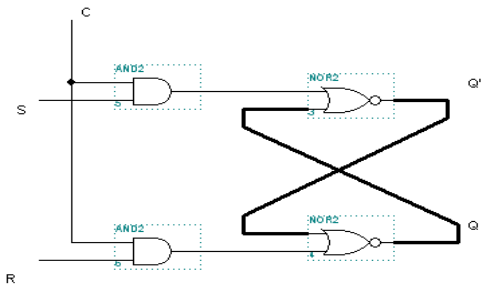


	R	S
تصادفی	0	0
نوشتن 0	1	0
نوشتن 1	0	1
حافظه ای	1	1

در هر دو حالت 0-0 را تصادفی نامیدیم زیرا چنانچه از این حالت به حالت حافظه‌ای برگردیم نتیجه معلوم نیست و می‌گویند کهRace پیش آمده است.

Flip Flop :

قادر به ذخیره سازی یک بیت اطلاعات است و برای هماهنگ سازی پالس ساعت نیز دارد.

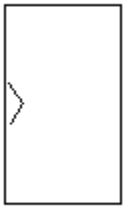


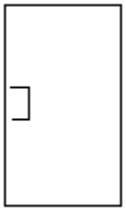
همانطور که مشهود است تنها زمانی مقدار نوشته می‌شود که $c = 1$ باشد.

در کل برای سادگی طراحی تغییرات المان‌های حافظه همزمان است.

فلیپ فلاپی که مدارش را در بالا دیدید حساس به سطح مثبت است، در کل بر اساس این نوع تقسیم بندی به شکل زیر می‌رسیم.

- فلیپ فلاپ
  - حساس به سطح
    - سطح یک (مثبت)
    - سطح صفر (منفی)
  - حساس به لبه
    - حساس به لبه بالا رونده(مثبت)
    - حساس به لبه پایین رونده (منفی)

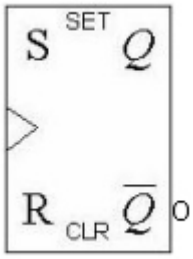




در شکل سمت راست فلیپ فلاپ حساس به سطح و در شکل سمت چپ فلیپ فلاپ حساس به لبه نمایش داده شده است.

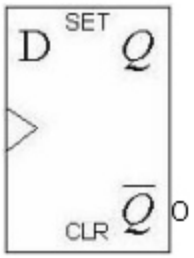
فلیپ فلاپ‌ها انواع مختلف دارند که به شرح زیر است:

RS-Flip Flop



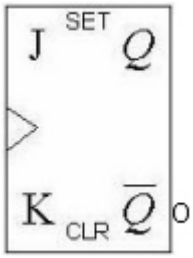
Inputs			Outputs		
S	R	C	Q	Q'	Comments
0	0	↑	Q	Q'	No change
0	1	↑	0	1	RESET
1	0	↑	1	0	SET
1	1	↑	?	?	Invalid

D-Flip Flop



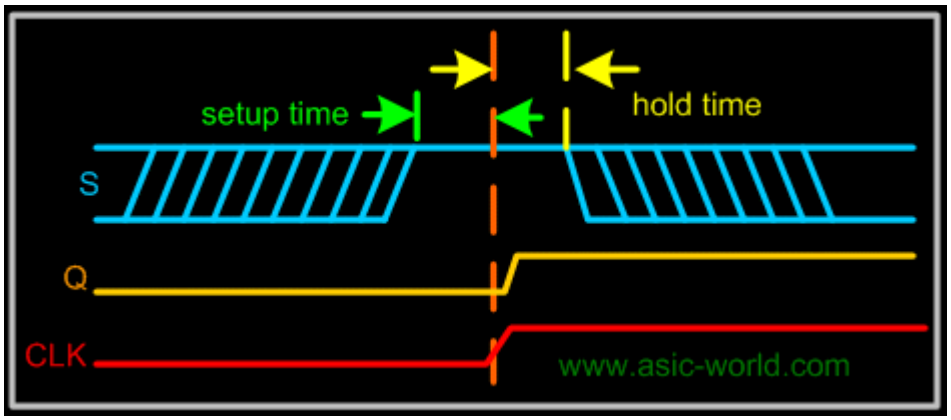
Inputs		Outputs		
D	C	Q	Q'	Comments
0	↑	0	1	RESET
1	↑	1	0	SET

JK-Flip Flop



Inputs			Outputs		
J	K	C	Q	Q'	Comments
0	0	↑	Q	Q'	No change
0	1	↑	0	1	RESET
1	0	↑	1	0	SET
1	1	↑	Q'	Q	Toggle

برای جلوگیری از حالت race و به وجود آمدن مقدار تصادفی برای پالس ساعت دو بازه‌ی زمانی t_s و t_h تعریف می‌شود که در این بازه مقدار ورودی فلیپ فلاپ نباید تغییر کند.



Setup time: کمترین بازه‌ی زمانی که مورد نیاز است ورودی قبل از گذار پالس ساعت پایدار باشد.

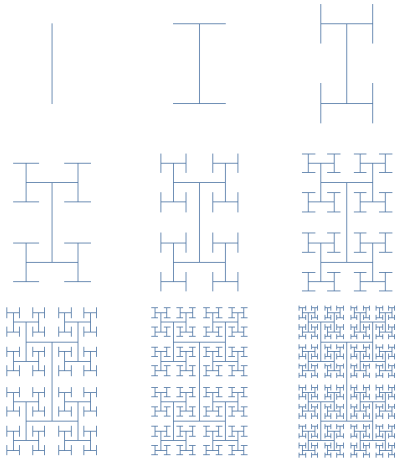
Hold time: کمترین بازه‌ی زمانی که مورد نیاز است ورودی بعد از گذار پالس ساعت پایدار باشد.

مقایسه‌ی مدارهای سنکرون و آسنکرون

مدارهای سنکرون	مدارهای آسنکرون
امکان قطع شدن مدار در آن وجود دارد. هم‌شنوایی[1] رخ می‌دهد. (به خاطر حجم بالای سیم‌ها) مصرف سیم بالا می‌رود. مشکل تاخیر وجود دارد. (سیم‌های نزدیکتر زودتر کلاک می‌خورند*) مدار گرم می‌شود. طراحی این مدارها نسبتا ساده است.	سرعت این مدارها بالاست. طراحی این مدارات دشوار است. توان مصرفی پایین.

* برای رفع مشکل تاخیر کلاک در مدارهای سنکرون از H-Tree استفاده می‌کنند.

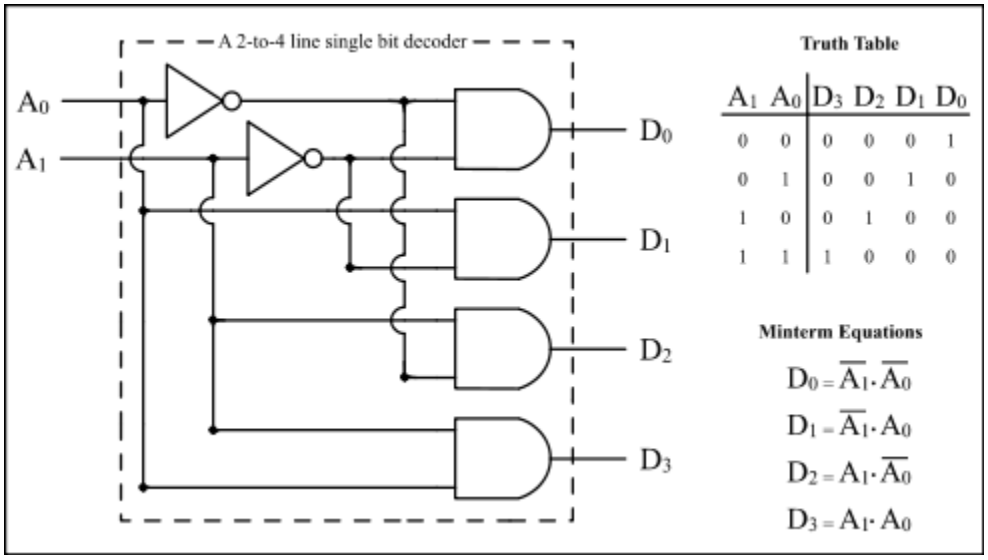
1 CROSS TALK، یعنی مقدار سیم‌های همسایه روی هم تاثیر می‌گذارد.



شکل 1 نمونه‌ای از H-Tree

Decoder

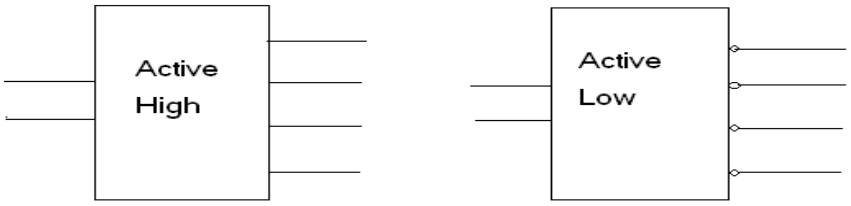
رمزگشـا: این گونه عمل می‌کند که n خط ورودی دارد و بر حسـب عدد ورودی یکی از 2^n خط خروجی(شماره ورودی) فعال شده و مابقی غیر فعال می‌شوند.



همیشه یک خروجی فعال و مابقی غیر فعال هستند، خروجی فعال خروجی است که کد آن در ورودی داده شده است.

Active High ⟶ یک خروجی یک و باقی صفر

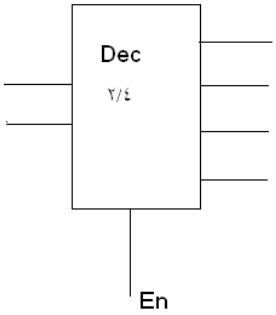
Active Low ⟶ یک خروجی صفر و باقی یک



با استفاده از Decoder (Active High) و گیت Or یا Decoder(Active Low) و گیت And هر تابع منطقی‌ای قابل پیاده سازی است.

Decoder with Enable Input

در این نوع Decoder خط ورودی En مشـــخص می‌کنـد که آیا خروجی‌ای فعال باشد یا خیر.



- $En = 0 \longleftarrow$ تمام خروجی‌ها غیر فعال
- $En = 1 \longleftarrow$ مانند Decoder عادی

Encoder

رمز کننده: در هر لحظه یک ورودی فعال است و مابقی غیر فعال و در خروجی کد شده‌ی ورودی فعال را خواهیم داشت.

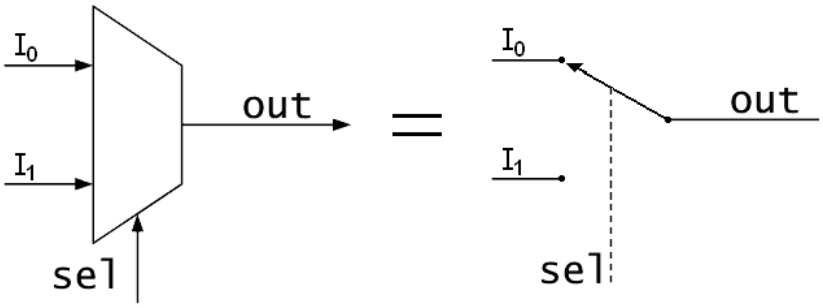
همانند Decoder دو منطق Active High و Active Low دارد.

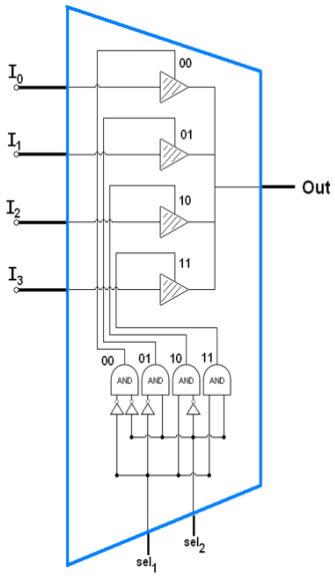
Encoder اولویت دار

برای ورودی‌هـا هم اولویـت قـائل می‌شـــویم، به این ترتیب دیگر لزومی ندارد که در ورودی تنها یک خط فعال باشـــد. در خروجی Encoder اولویت دار چنانچه هیچ یک از ورودی‌ها فعال نباشـــند در خروجی خط z فعال می‌شود.

MUX

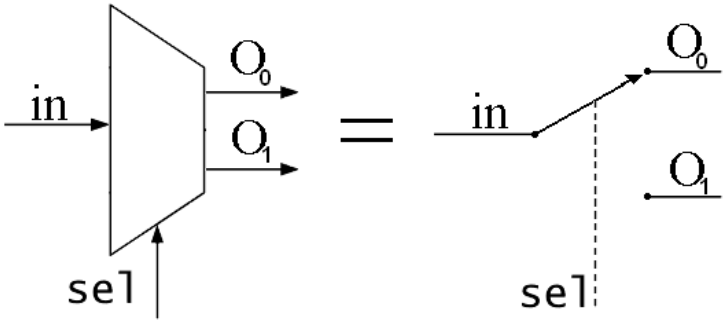
تسـهیم کننده: 2^n خط ورودی و n خط انتخاب دارد و یک خط خروجی، بنا بر ورودی انتخاب خط ورودی را به خروجی منتقل می‌کند.

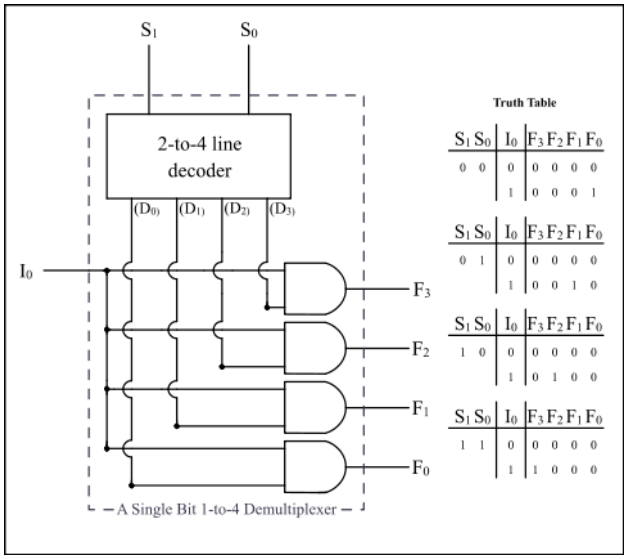




Demux

در حقیقت همان Decoder با ورودی Enable است.



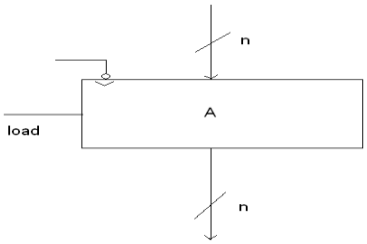


سوال: تفاوت Demux و Decoder در چیست؟

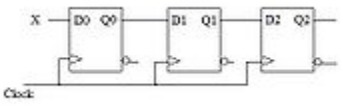
البته تفاوت‌های زیادی ممکن است به نظر برسد اما آنچه اینجا می‌خواهیم بگوییم این است که در Demux یک خروجی فعال و مابقی Z هستند اما در Decoder یک خروجی فعال و بقیه غیر فعال هستند.

Register

ثبات: گروهی از فلیپ فلاپ‌ها به عنوان مجموعه‌ی واحد می‌باشند که n بیت را ذخیره می‌کنند.



Parallel in-Parallel out



Shift Register

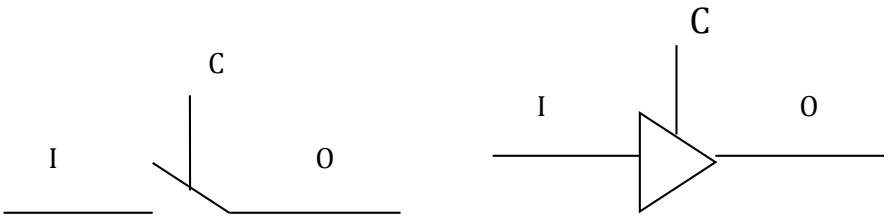
و از انواع دیگر می‌توان به Serial in-Serial out و Rotate اشاره کرد.

- در طراحی‌ها ثبات بدون Load نباید داشته باشیم.
- اگر ثبات output enable باشد می‌تواند خروجی‌ها را HighZ یا فعال کند. به این ترتیب کار MUX را هم می‌تواند انجام دهد.
- در لحظه‌ی بالارونده‌ی کلاک داریم:

$$\begin{cases} if\ load = 0 \rightarrow O = Z \\ if\ load = 1 \rightarrow O = 1 \end{cases}$$

Tri-State Buffer

برای اتصال خروجی‌ها به هم از Tri-State یا MUX استفاده می‌کنیم. خروجی این قطعه میتواند علاوه بر دو حالت 0,1 دارای حالت سومی باشد که عملا بصورت امپدانس بالا و یا حالت قطع عمل میکند.



$$\begin{cases} if\ c = 0 \rightarrow O = Z \\ if\ c = 1 \rightarrow O = 1 \end{cases}$$

RANDOM ACCESS MEMORY (RAM)

شاید بهتر بود نام این حافظه را Direct Access Memory می‌گذاشتند چرا که می‌توانیم با داشتن آدرس هر خانه‌ی حافظه به طور مستقیم به محتویات آن دسترسی پیدا کنیم. این حافظه از تعدادی خانه یا سلول تشکیل شده است و هر خانه، قابلیت نگهداری یک داده را دارد. هریک از این خانه‌ها با آدرسی منحصر به فرد مشخص می‌شود. آدرس اولین خانه حافظه، صفر است و آدرس هر خانه، یک واحد از خانه‌ی قبلی‌اش بیشتر است، هر آدرس حافظه، قابلیت نگهداری یک یا چند بایت را دارد.

شکل 2 RAMها

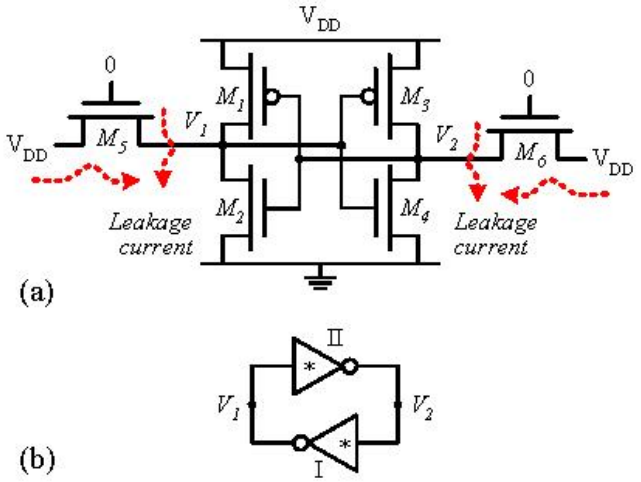
داده‌های موجود در RAM قابل پاک شـدن و جایگزینی با داده‌های دیگر هستند و هر نوع وقفه‌ای در جریان برق رایانه، موجب از بین رفتن داده‌های موجود در RAM می‌شـود. البته در نوع خاصـی از RAMها قابلیت نگهداری داده برای زمان طولانی‌تر وجود دارد. استفاده از این نوع حافظه‌ها، برای نگهداری موقت اطلاعات تا زمان پردازش یـا انتقال نتایج به بیرون از رایانه و یا ذخیره در حافظه‌های جانبی اسـت. داده‌های مورد نیاز پردازنده ابتدا وارد RAM شـده و سـپس پردازش روی آنها صـورت می‌گیرد. به RAM، حافظه خواندنی و نوشـتنی (RWM) هم میگویند.

از نظر تکنولوژی ساخت، دو نوع RAM وجود دارد:

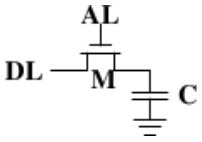
1. Dynamic RAM (DRAM)
2. Static RAM (SRAM)

DRAM نسـبت به SRAM دارای سـرعت دسـترسـی پایین‌تر و هزینه‌ی سـاخت کمتر اسـت. در این نوع حافظه اطلاعات باید به طور مرتب تجدید شـوند وگرنه از بین خواهند رفت (البته این کار به صـورت خودکار صـورت می‌گیرد). از DRAMها در سـاخت حافظه‌ی اصـلی اسـتفاده می‌شود. به خاطر هزینه‌ی بالای SRAM معمولا در حافظه‌ی نهان از آن استفاده می‌شود و حافظه‌های با حجم بالا معمولا DRAM هستند.

ساختار داخلی SRAM و DRAM در شکل‌های زیر آمده است.



شکل 3 ساختار داخلی SRAM



شکل 4 ساختار داخلی DRA

در جدول زیر مقایسه‌ی این دو نوع RAM آمده است.

جدول 1 مقایسه‌ی SRAM , DRAM

RAM	معایب	مزایا
DRAM	نیاز به Refresh دارد توان مصرفی بالا سرعت پایین	هزینه‌ی کم چگالی بیتی بیشتر
SRAM	هزینه‌ی زیاد چگالی بیتی کمتر	توان کم سرعت بالا نیاز به Refresh ندارد

- SDRAM (Synchronous DRAM) نوعی از DRAM است که با کلاک پالس Refreshمی‌شود.
- DDR RAM (Double Data Rate RAM) نوعی از RAMاســـت که هم در لبه‌ی بالارونده و هم لبه‌ی پایین‌رونده Read, Write می‌کند.

ROM

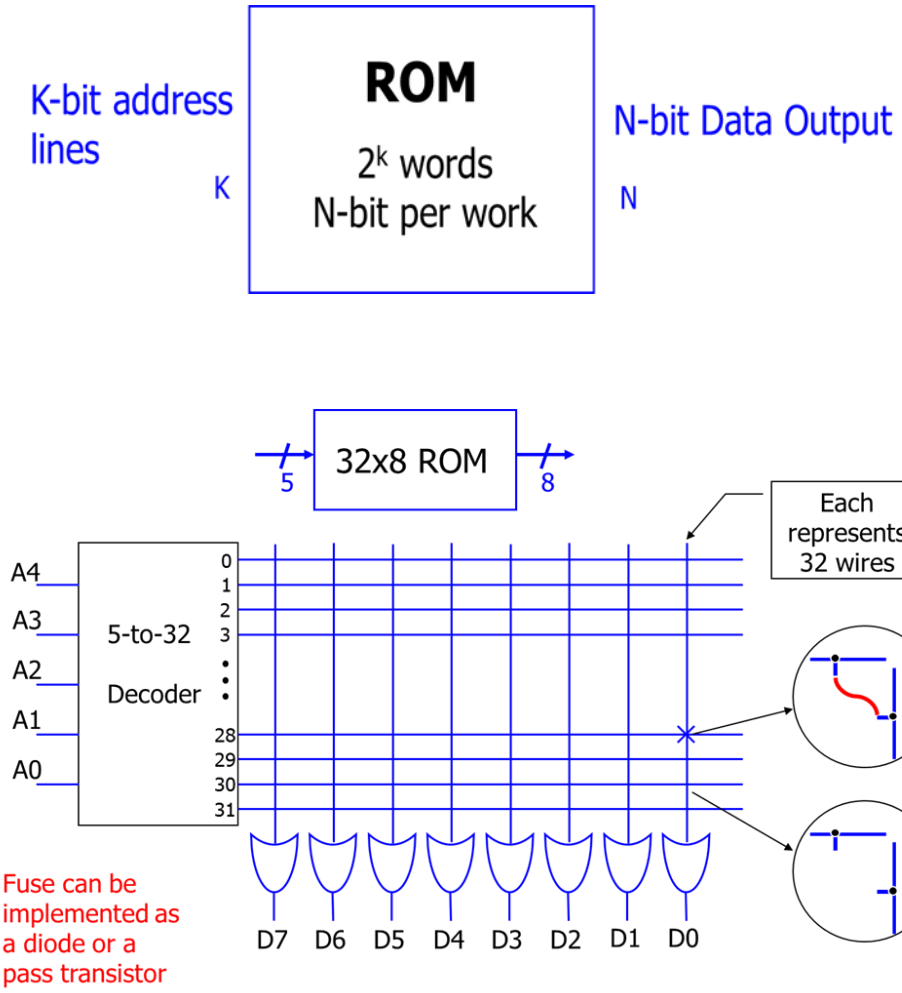
حافظه‌ای اســت فقط خواندنی که محتوی آن یکبار نوشــته شــده و پس از نصــب در کامپیوتر تغییری در آن داده نمی‌شود.

معمولا از این حافظه برای ذخیره برنامه هائی نظیر bootstrap loader که برای راه اندازی اولیه کامپیوتر مورد نیاز هستند استفاده می‌شود.

این حافظه انواع مختلفی دارد:

- **PROM**: ROMهایی که هنوز برنامه نویسی نشده و تنها یک بار می‌توان روی آن نوشت.
- **EPROM**: PROMهایی که قابلیت پاک کردن هم دارند(با استفاده از اشعه ماوراء بنفش)
- **EEPROM**: برای پاک کردن نیاز به ماوراء بنفش نیست و با برق پاک می‌شود.

اطلاعات باینری بطور دائمی در حافظه ذخیره می‌شوند و با قطع برق از بین نمی‌روند.

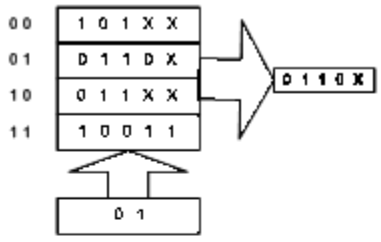


در این شکل یک ROM با ابعاد 8×32 آورده شده است که برای برنامه نویسی می‌بایست فیوز خط مربوطه را بسوزانیم.

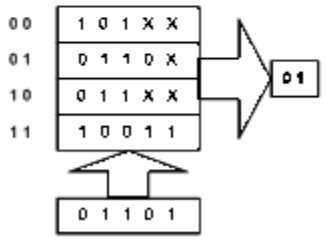
Content Addressable Memory (CAM)

تا به حال به طرز کار حافظه‌ی انسان دقت کرده‌اید؟ اغلب با دیدن یک تصویر ناقص، بلافاصله کامل آنرا به خاطر می‌آورید، یا با دیدن تصویر یک شخص سریعا نام او را می‌گویید، یا با خواندن یک متن سریعا تمامی مطالب مربوط به آن را به ذهن می‌آورید. در واقع ذهن انسان یک نوع حافظه‌ی آدرس‌دهی شده بر اساس محتواست (Content Addressable Memory). همانگونه که از این نام مشخص است در این نوع حافظه، با دادن محتوای یک خانه از حافظه، بلافاصله آدرس آن به عنوان خروجی داده می‌شود. یکی از مهم‌ترین تفاوت‌های حافظه انسان با حافظه کامپیوتر در نوع آدرس دهی است. در حافظه کامپیوتر اساس کار بر پایه آدرس خانه‌های حافظه یا آدرس اطلاعات بر روی حافظه دائم است. به عنوان مثال برای دستیابی به یک تصویر یا متن خاص، باید آدرس حافظه یا فایل مربوط به آن تصویر یا متن را داشته باشید. اما با داشتن خود تصویر یا متن نمی‌توانید به سادگی

آدرس حافظه مربوطه را بیابید. اینجا بود که ایده‌ی ســـاخت حافظه‌هایی بوجود آمد که بتوانند بر اســـاس محتوا جستجو کنند.

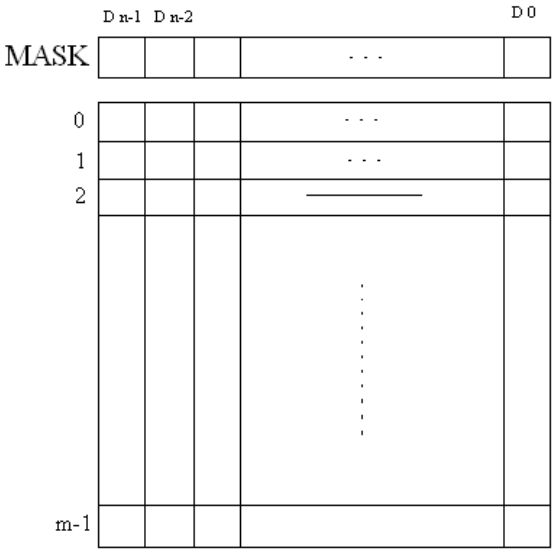


حافظه های آدرس پذیر



حافظه های **CAM**

همانطور که در شکل ملاحظه می‌کنید در حافظه‌های قدیمی با دادن آدرس، محتوای آدرس را دریافت می‌کردیم در حـالیکـه در حـافظه‌ی CAM با دادن محتوا آدرس داده‌ی مشـــابه با داده‌ی ورودی را پیدا می‌کنیم. طرز کار حافظه‌ی CAM به این صورت است که داده‌ی ورودی همزمان با تمام اطلاعات موجود در حافظه مقایسه می‌شود و اگر خود داده در حافظه وجود داشت، می‌گوییم Match رخ داده است.



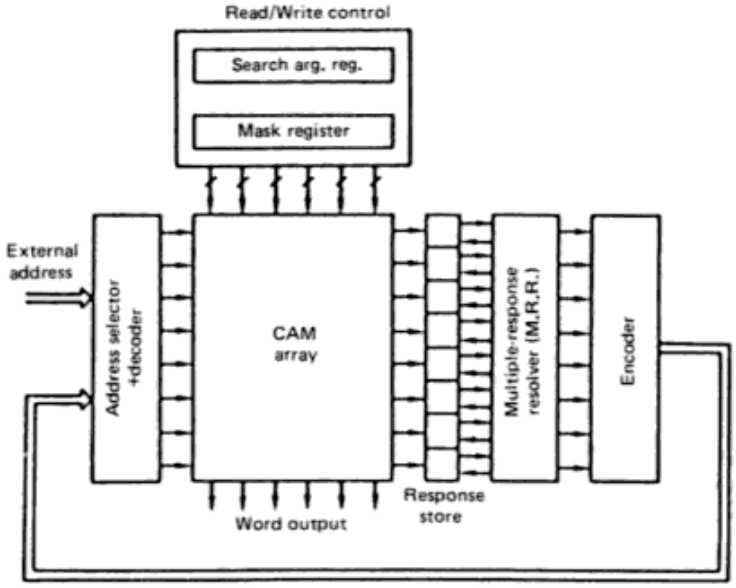
شکل 5 حافظه‌ی CAM

شکل 5 حافظه‌ی CAMای را نشان می‌دهد که طول داده‌ی ورودی آن n باشد و در حافظه m کلمه داشته باشد. بیت iام ورودی با بیتهای iام تمام کلمه‌ها XOR می‌شـود. به وضـوح کلمه‌ای مورد نظر ماسـت که نتیجه‌ی این XOR برای تمام بیتهای آن صفر باشد. یعنی در واقع mn مقایسه کننده داریم. این تعداد مقایسه‌گر توان مصرفی را به شدت بالا می‌برد. به طوری در مورد مقایسه‌ی CAM و سایر حافظه‌ها می‌توان گفت :

توان مصرفی آدرس پذیرها <توان مصرفی CAM

مساحت[۲] آدرس پذیرها $\sqrt{2}$ ≈ مساحت CAM

همانطور که گفته شـد پس از XOR، برخی از کلمات کاملا با ورودی برابر می‌شـوند که در این حالت می‌گوییم Match رخ داده است. به حالتیکه در آن بیش از یک Match رخ دهد Multiple Match می‌گوییم. دراین حالت با تابعی مشخص یکی از این نتایج برگردانده می‌شود.



شکل 6 نمای کلی CAM در سیستم

2 هزینه‌ی سخت افزاری یا HWCost معمولا به هزینه یا مساحت تعبیر می‌شود. مساحت در واقع متناسب با تعداد ترانزیستورهاست.

Verilog

کامپیوتر را به سطوح تجریدی تقسیم می‌کنند.

سه زبان برای سخت افزار داریم :

System C, VHDL, Vrilog

به جند دلیل از Verilog استفاده می‌کنیم :

Keywordهای سخت افزاری بهتری دارد.

سطوح تجرید مختلف را پشتیبانی می‌کند.

وقتی با Verilog برنامه نویسی می‌کنیم فایلی با پسوند v ساخته می‌شود که محتوایش توصیف سخت افزار است و اصطلاحا می‌توان آن را شبیه سازی کرد. برای شبیه سازی باید از سیمولاتور استفاده کنیم. ما از Modelsim استفاده می‌کنیم.

توجه کنید که در اینجا چیزی به نام برنامه نداریم بلکه در واقع ما یک توصیف می‌نویسیم.

هر کد Verilog با یک module شروع می‌شود. ماژول‌های تو در تو نداریم. اما می‌توان در یک ماژول، ماژول دیگری را instantiate کرد. در واقع ماژول‌ها componentهای سخت افزاری اند.

Verilog به C نزدیک است، برای مثال Verilog هم مانند C حساس به حروف کوچک و بزرگ است. در Verilog هر جمله یا statement باید به ’;‘ ختم شود غیر از endmodule

کدهای HDL کدهای parallel یا موازی هستند و نه sequential. یعنی همه‌ی مولفه‌ها مستقل از هم و موازی با هم کار می‌کنند. پس ترتیب مهم نیست و همه‌ی جملات موازی با هم اجرا می‌شوند.

به عنوان مثال:

```
module
      ---------------
      Declarations
      ---------------
      Parallel Statements
      ---------------
endmodule
```

قسمت اول در واقع المان‌های مولفه مثل سیم و گیت‌ها و. .. خواهند آمد مثل

reg , wire , parameter , input , output , task , function , …. .

در واقع یک سری سمبل تعریف می‌کنیم تا در قسمت statement از آنها استفاده کنیم.

بسته به نوع سطح تجرید statementهای متفاوتی خواهیم داشت.

Parallel statementها چند نوع دارند:

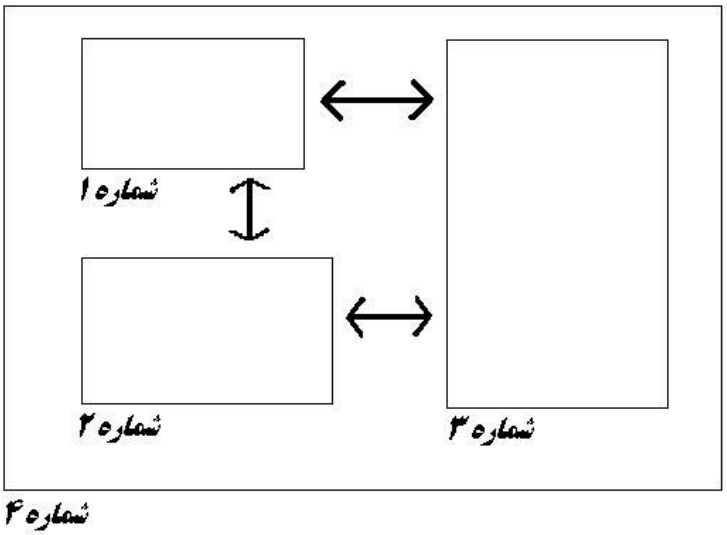
1. Behavioral
   a. Initial statement
   b. Always statement
2. Module instantiation
   a. برای شیئ گرفتن از ماژول‌های دیگر
3. Gate instantiation
4. UDP instantiation(User Defined Primitive)
   a. Gate Level
   b. Switch Level

هر گاه احساس کنیم که یک Basic Element در خود verilog نباشد می‌توان با UDP آن را تعریف کرد.

5. Continues Assignment یا RTL(Register Transform Level/Language)

بعضی از شرکت‌هایی که FVGA و ICهای برنامه پذیر تولید می‌کنند، همراه IC، HDL و سیمولاتور هم تولید می‌کنند. البته نرم افزارهای ویژوال برای تولید کد هم وجود دارد.

بهتر است همیشه برای نوشتن VHDL دیاگرام بکشیم مثلا در شکل زیر 4 تا ماژول باید تعریف کنیم. ابتدا ماژول‌های 1 و 2 و 3 و در سپس شماره 4 که باید در آن 1 و 2 و 3 را instantiate کنیم.



چند نکته :

1. UDP خاص کاربر است.
2. در واقع خود UDPو گیت‌ها هم نوعی ماژول هستند.
3. تمام مولفه‌ها به هم وصلند و با هم کار می‌کنند و هر مولفه به شرط ورودی گرفتن خروجی می‌دهد.
4. مدارها باید پایدار باشند که وقتی ورودی گرفت بالاخره به یک حالت stable برسد.

معرفی کلمات کلیدی Verilog :

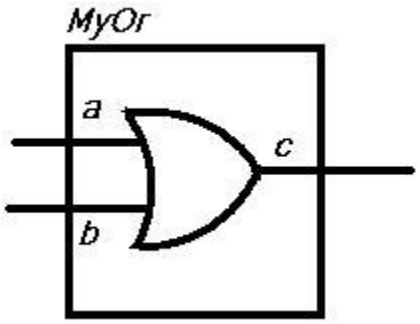
- reg یعنی رجیستر می‌خواهیم.
- Wire یعنی سیم می‌خواهم.

اما چرا از اینها استفاده می‌کنیم؟

اگر نگوییم سیم داریم هم کد کار می‌کند. در واقع این برای شبیه ساز است تا سمبول‌ها و مولفه‌های داخل کد را بفهمد و بتواند برای خودش Symbol Table بسازد.

مثال :

توصیف ماژول MyOr :



بعد از کشیدن شکل می‌بایست سطح تجرید را مشخص کنیم. (همیشه با سطح گیت کار می‌کنیم مگر اینکه ذکر کنیم با RTL)

رسم است که ابتدا خروجی ماژول را بنویسند

```
module MyOr(c, a , b)
       Output c;
       Input a, b;
       …
endmodule
```

حالا به جای سه نقطه در چند سطح تجرید می‌نویسیم.

1. سطح گیت :

اسم ماژول هم اهمییتی ندارد و بیشتر برای سیمولاتور است.

```
MyOr mo(c, a , b);
```

2. سطح RTL :

این همان مفهوم لحیم کاری در سخت افزار است.

```
assign c = a|b;
```

عبارت بالا یک Continues Assignment است.

3. سطح Behavioral:

Always statement همیشــه استفاده و اجرا می‌شود اما initial statement یعنی فقط در t=0 اجرا شود که برای initialize کردن ورودی‌ها به کار می‌رود.

Always به تنهایی خوب نیســـت چون ســـیمولاتور را مشـــغول می‌کند برای همین sensitivity list جلوش می‌گذارند (با پرانتز) که هر وقت تغییر کردند، اجرا شود و مثلا در اینجا هر وقت aیا b تغییر کند:

```
always @ (a or b)
	begin
		c = a|b;
	end
```

عبارات بین begin و end به صـورت ترتیبی اجرا می‌شـوند. دسـتورات داخل این بلوک نیاز به assign ندارند و عملا دسـتورات نرم افزاری اند. توجه کنید که متغییرهایی که در سـمت چپ قرار می‌گیرند باید holder باشند. سیم holder نیست پس در اینجا یک رجیستر هم سر راهش می‌گذاریم یعنی :

```
module myor(c , a , b)
	output c;
	input a, b ;
	reg c;
always @ (a or b)
	begin
		c = a|b;
	end
endmodule
```

در Verilog می‌توان Hierarchical Abstract Level داشت و چند تا سطح تجرید هم می‌توان با هم گذاشت.

دقت کنید که سر ورودی و خروجی رجیستر گذاشتیم و اینها می‌توانند دوباره تعریف شوند.

اگر اسم رجیستر را عوض کنیم مثلا x میگوییم:

```
always @ (a or b)
begin
	x = a|b
end
assign c=x;
```

یک سیم یا input یا output و یا inout است.

مثال Full Adder:

```
module fulladder(s, x, a, b, c)
	output s, x;
	reg s, x;
	input a, b ,c;
	always @ (a or b or c)
	begin
		s = a ^ b ^ c;
		x = (a & b) | (b & c) | (c & a);
	end
endmodule
```

RTL:

```
assign x = (a & b) | (b & c) | (c & a);
assign s = a ^ b ^ c;
```

در این حالت RTL، به صورت موازی اجرا میشود. هرگاه عبارات سمت راست تغییر کنند، دستورها موازی با هم اجرا می‌شوند.

Gate Level:

```
and a1 (w1, b, a);
and a2 (w2, b, c);
and a3 (w3, a, c);
or o1(x, w1, w2, w3);
```

میتوان گیت‌ها را دوورودی هم گرفت. در اینجا w1هم ایجاد شد که اینها را باید در Declaration بیاوریم.

```
xor x1(s, a, b, c);
output x, s;
input a, b, c;
wire w1, w2, w3;
```

میتوان خط فیدبک هم به همین صورت تعریف کرد.

- در سطح گیت نیازی به assign نیست. Assign فقط در RTL استفاده میشود.
- دقت میکنیم که هر چه به سطوح پایین تر میرویم، جزئیات بیشتری را میبایست توصیف کرد و توصیف طولانی تر میشود. برعکس هر چه به سطوح بالاتر میرویم، توصیف راحت تر است.

تمرین: مدار هر یک از موارد زیر را در سطح گیت توصیف کنید. با استفاده از TestBench در محیط ModelSim شبیه سازی کنید.

الف) Decoder 2->4

ب) Encoder 8->3

ج) Mux 4X3 -> 1X3

د) DeMux 1->8

ه) 4 bit comparator

Module Instantiation:

مثال:

```
module x;
.
.
.
myor mo1(x, y, z);
.
.
.
```

گاهی به ازای هر ماژول توصیف شده یک ماژول تست بنچ برای آن مینویسم که نقش Input Generator را برای آن ایفا کند. در ModelSim هم میتوان اینطور ورودی داد و هم میتوان سیگنال‌ها را تک تک وارد کرد.

Test Bench: مداری که مدار دیگر را تست میکند. بعضی از نرم افزارها مانند ModelSim قابلیت ایجاد این مدارها را در خود دارند.

کار بهتر این است که یک ماژول جدا برای TestBench نوشته شود. مثلا TestBench برای FA:

```
module fa(s, x, a, b, c);
.
.
.
endmodule;
module tb (a, b, c);
output a, b, c;
reg a, b, c;
initial
        begin
                a = 1;
                b = 0;
                c = 1;
        end
endmodule
```

اما این دو را باید به هم متصل کرد. در ادامه‌ی قبلی میگوییم:

```
module tester:
        wire p, q, r, m, n;
        fa mfa (m, n, p, q, r);
        tb mtb (p, q, r);
endmodule
```

برای اینکه به عنوان ورودی به تابع دیگر به ترتیب دلخواه خودمان بدهیم، از call by name استفاده میکنیم.

مثلا

```
tb mtb (. c( r) ,. b(q),. a(p))
```

با ModelSim موقع شبیه سازی کردن میپرسد که top Module چیست. مثلا در اینجا Tester است. البته در این مورد خودش متوجه میشود وگرنه اگر ماژول‌ها هم سطح بودند، میپرسید.

تذکر: سایت www. opencores. com کدهای توصیفی open دارد.

اگر بخواهیم ورودی‌های مختلف و با حالات مختلف بدهیم، میتوانیم از تاخیر(#) استفاده کنیم، یعنی:

```
module tb (a, b, c);
.
.
.
        c =1;
        #5 c=0;
        .
        .
        .
```

این یعنی c را 5 واحد زمانی دیرتر مساوی صفر قرار دهد و همینطور میتوان مقدار داد یعنی:

```
#5 c=0;
#5 b=0;
```

وقتی #5 رسید یعنی 5 واحد صبر کن چرا که begin و end حتما sequential اجرا میشود. مثلا بگوییم:

```
#5 c=0;          t=5
#5 b=0;          t=10
C=1;             t=10
#15 a=0;         t=25
```

پیش فرض واحد زمانی 1 ns است اما directive دارد. همه directiveها با ‘ شروع میشود. مثلا میگوییم:

```
‘Timescale      1ns/100ps (دقت)
‘Timescale      1ns
```

Directiveهـای مختلفی وجود دارد. مثلا undefined var یعنی چیزهـایی کـه تعریف نکردیم را مثلا از نوع wire تعریف کنیم و یا:

```
‘Include A. v
```

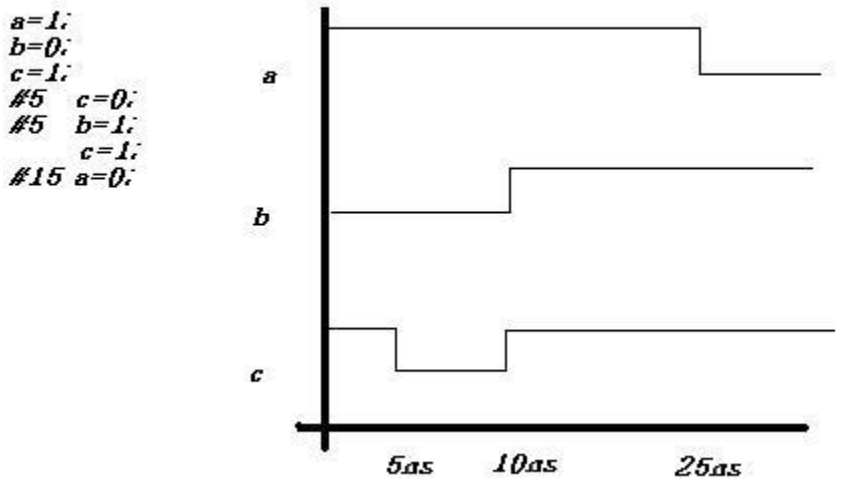
گاهی برای تاخیر میخواهیم نایستد. یعنی ;c=0 #5 باشد اما دستورات بعدی را همان لحظه ادامه دهد.

دراینجا دو نوع assignment داریم:

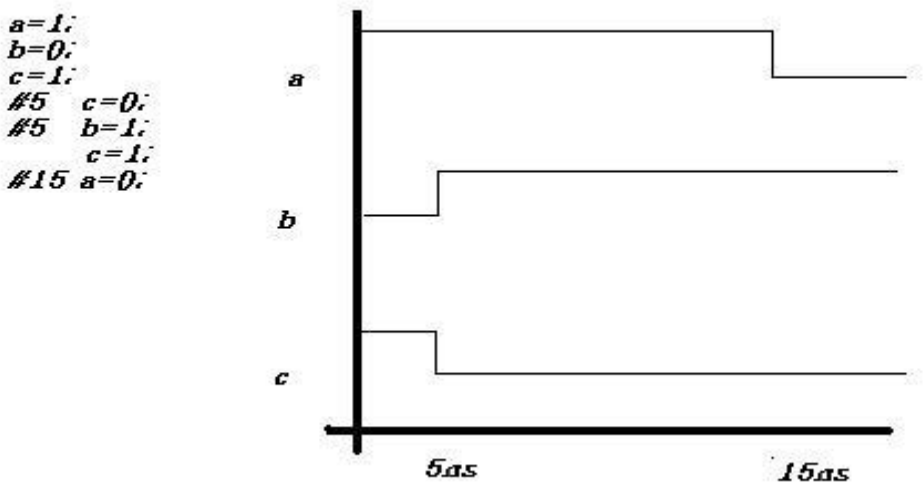
1. Blocking assignment
2. Non-blocking assignment

مثلا در دستورات قبلی از نوع اول است که سیمولاتور می‌ایستد. برای حالت دوم میگوییم:

#5 b<=0;



حال فرض کنید به جای تمام =های بالا، <= بود:



مثلا اگر به جای خط پنجم b<=1 باشد، زمان‌ها 5و 10و25 میشود. برای <= میتوان # را برداشت. = یعنی این دستورات را انجام بده بعد سری بعدی‌ها را انجام بده.

a=1	a<=1
b=a&b	b<=a&b

در اینجا در مورد <=ها، دستورات همزمان انجام میشوند یعنی لزما a=1 نیست، با a قبلی و b عمل & انجام میشود و همزمان 1 به a و a&b به b میرود.

علامت <= یعنی موازی باشد و طبیعت سخت افزاری بدهیم.

دقت کنیم که دستورات دنباله‌ای انجام میشوند.

اگر هم c=0 باشد و هم c=1 آنگاه c=x میشود اما در طراحی‌ها نباید این مورد را داشته باشیم(یعنی x و z نباید داشته باشیم)

نکته: اینکه تاخیرها را هنوز سخت افزاری پیاده سازی نکرده اند و چنین کدهایی فقط در سطح شبیه سازی‌اند.

فرض کنید نیاز داریم چند خط که دنباله‌ای هستند را به طور موازی اجرا کنیم. میتوان وسط قسمت دنباله‌ای، fork - join باز کنیم تا موازی اجرا شوند.

```
begin
        ...
        fork
                ....
        join
        ....
end;
```

دستورات داخل fork-join موازی اند و <= نیز برای موازی اجرا کردن است ولی فقط برای assignment ولی دستور fork join برای assignment نیز هست.

تعریف حافظه:

reg a; → 1 bit

reg pc [32:1]; →

32	31	30					...	2	1

reg pc[1:32]; →

1	2	3					...	31	32

حافظه، آرایه‌ای از

ثبات هاست. مثلا 1KB یعنی 1024*8 را انگونه میگوییم:

```
reg [0:1023] mem [7:0] ;
```

یعنی از سلول [7:0] به اندازه 0:1023 تکرار کن، یعنی

7	6	5	4	3	2	1	0
.							
.							
.	ردیف 1023ام						

دسترسی به بیت ها:

```
reg pc [0:31];
 pc [5];
reg pc [48:17]
 pc [10]; → syntax error
reg [0:1023] mem [7:4];
```

یعنی برای دسترسی به سطر چهارم، اول کل 7 بیت را در یک رجیستر بگذاریم یعنی:

```
reg temp [7:10]
...
temp= mem [4];
temp [6]; → دسترسی به سطح چهارم و بیت ششم
```

حال فرض کنید بخواهی بیت ششم آرایه‌ی mem را بگریم. Part selection هم داریم. مثلا

```
reg a [48:17];
temp= a [48:40]; → part selection
```

اما برای بیت ششم آرایه و یا ساختن یک رجیستر مجازی:

```
reg temp [7:0];
...
temp= {pc [43,40] , a, pc[12,10]} + 10
ویا:
{. .. } = {. .. } + 10
```

مثلا میتوان گفت pc[40,43] و. ..

برای اعداد حالت عادی دهدهی است:

دودویی: b'11101101

جداسازی: b'1110_1101 → "_" نوشته میشود ولی فقط برای خوانایی است

برای for و. .. باید int تعریف کنیم. میتوان در تعریف نوشت:

```
integer i;
assign w1 = {p [31,1] , a}
```

سلسله مراتب حافظه

فرض کنید مدیرعامل یک شـرکت سخت‌افزاری ساخت قطعات کامپیوتری هستیم و می‌خواهیم محصولی را ارزه کنیم که هم از نظر کارایی خوب باشـد و هم قیمت بالایی نداشـته باشـد. برای دسـتیابی به این مهم باید دو

محدودیت متضاد را در نظر بگیریم: پول(یا به تعبیری دیگر هزینه‌ی سخت‌افزاری) و کارایی(سرعت و حجم). باید سعی کنیم با حداقل پول به بیشترین کارایی برسیم.

طبیعتاً در مورد حافظه‌ها هم دو عامل پول و کارایی مهم‌اند. از این حیث می‌توان حافظه‌ها را دسته بندی کرد:

1. Flip Flop
2. Register
3. Register File: در طراحی CPU ممکن است از تعداد زیادی رجیستر استفاده کنیم. بدین ترتیب بهتر است آنها را سازماندهی کرده و هر چندتایی (حداقل 8 و حداکثر 256تا) را در یک دسته قرار دهیم. هر کدام از این دسته‌ها را یک Register File\Bank می‌گویند.
4. Cache
5. Main Memory
6. Magnetic Disk: مثل هارد و فلاپی و. ..
7. Optical Disk: مثل CDها
8. Tapeها و کارت پانچ‌ها: Tapeها برای ذخیره‌ی انبوهی از اطلاعات و به عنوان آرشیو مورد استفاده قرار می‌گیرند. تنها ضعف آنها ترتیبی[3] بودن دسترسی و لذا سرعت کم دسترسی به اطلاعات آنهاست. البته این نوع حافظه‌ها انتها ندارند و این خود می‌تواند یک مزیت باشد.

در این تقسیم بندی از بالا به پایین سرعت کاهش یافته اما در عوض قیمت و حجم(ظرفیت) افزایش می‌یابد.

اما فرض کنیم بخواهیم یک کامپیوتر را با تمام متعلقات آن به مشتری ارزه کنیم. به نظر شما جه نوع حافظه‌ای را در این کامپیوتر قرار دهیم؟ اگر کل حافظه‌ی آن از نوع Flip Flop باشد یا کل آن از نوع Tape باشد، خوب است؟ عامل پول را چطور در نظر بگیریم؟ بهترین کار آن است که از هر نوع حافظه درصدی در کامپیوترمان داشته باشیم.

اگر حجم حافظه استفاده شده از سطح i ام را در این کامپیوتر خیالی با c_i نشان دهیم، داریم:

3 Sequential

$c_1 < c_2 < \cdots < c_8$

اگر زمان دسترسی[۴] مؤلفه‌ی i ام را با d_i نشان دهیم:

$d_1 < d_2 < \cdots < d_8$

که البته به طور تقریبی:

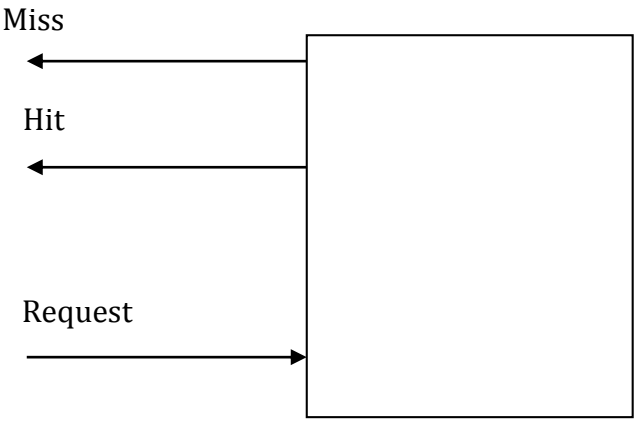
$d_1 \cong 4ns$

$d_4 \cong 10ns$

$d_6 \cong ms$

$d_8 \cong s$

همانطور که گفتیم در طراحی یک کامپیوتر تمام این سطوح وجود خواهند داشت، اما مکانیزم دسترسی به اطلاعات در این طراحی به گونه‌ای است که تا آنجا که بتوانیم داده‌ها را در سطوح بالا نگه‌داری می‌کنیم و در صورتی که به داده‌ای نیاز داشته باشیم، از بالاترین سطح شروع کرده و در صورتی که داده موجود بود (hit) که هیچ و اگر نبود(miss) به سطح پایین‌تر می‌رویم. به این ترتیب چون مطمئنیم داده‌ی مورد نیاز لااقل در پایین‌ترین سطح موجود است، ختماً در مرحله‌ای داده را خواهیم یافت. شمای کلی کار با هر مؤلفه‌ی حافظه در زیر آمده است.



برای یک مؤلفه‌ی خاص، بهترین حالت آن است که همه‌ی درخواست‌ها hit شوند. پس برای ارزیابی کارایی یک مؤلفه Hit Ratio را به صورت زیر تعریف می‌کنیم:

4 Access Time

$$\text{Hit Ratio} = \frac{\#\text{hits}}{\#\text{hits} + \#\text{misses}}$$

هدف آن است که سطوح بالای حافظه را طوری طراحی کنیم که Hit Ratio را برای آنها بالا ببریم. در بادی امر این موضوع چندان ممکن به نظر نمی‌رسد اما کاربر کامپیوتر یک انسان است؛ انسانی که ذهنی ساخت‌یافته دارد و نحوه‌ی فکر کردن او به مسائل دارای نظم خاصی است. به همین دلیل می‌توان حافظه‌هایی ساخت که Hit Ratio بالایی دارند. مثلاً این نسبت در حافظه‌های نهان[5] بیش از 94 درصد است!

فرض کنید Hit Ratio مؤلفه‌ی حافظه در سطح i ام را با h_i نشان دهیم. در واقع می‌توان h_i را احتمال حضور داده در سطح i ام دانست. بدیهی است که $h_8=1$. با این توصیف، به راحتی و با استفاده از مفاهیم امید ریاضی، می‌توان رابطه‌ی زیر را برای متوسط زمان دسترتی به داده به دست آورد:

$$Average\ Access\ Time = h_1 d_1 + (1-h_1)(h_2 d_2 + (1-h_2)(\cdots (h_{n-1} d_{n-1} + (1-h_{n-1}) d_n) \cdots))$$

اما چرا در پرانتز دوم $h_2 d_2$ گذاشتیم و چرا به جای آن $h_2(d_1 + d_2)$ ننوشتیم؟ مگر نه اینکه در صورت miss شدن درخواست به حافظه‌ی سطح اول حداقل به اندازه‌ی d_1 زمان صرف شده و بعد از صرف این زمان به سراغ سطح دوم می‌رویم؟ اگر بخواهیم همان $h_2 d_2$ را در رابطه قرار دهیم به این معنی است که برای بدست آوردن هرداده باید به همه‌ی مؤلفه‌های حافظه درخواست دهیم که در این صورت توان مصرفی بالا و همچنین سیم‌کشی اضافه‌تری خواهیم داشت که اصلاً خوب نیست.

حقیقت این است که رابطه‌ی گفته شده، همان رابطه‌ای است که معمولاً برای بدست آوردن متوسط زمان دسترسی مورد استفاده قرار می‌گیرد. در واقع علت استفاده نکردن از $h_2(d_1 + d_2)$ آن است که معمولاً تکنولوژی حافظه در سطوح مختلف متفاوت است و لذا $d_{i-1} \ll d_i$ و لذا d_{i-1} نسبت به d_i قابل صرف نظر کردن است.

5 Cache

مثال: فرض کنید در کامپیوتری فقط دو نوع حافظه‌ی نهان و حافظه‌ی اصـــلی وجود دارد که زمان دسـترسی آنها به ترتیب $10ns$ و $1\mu s$ باشد. متوسط زمان پاسخ به درخواست یک داده چقدر است؟

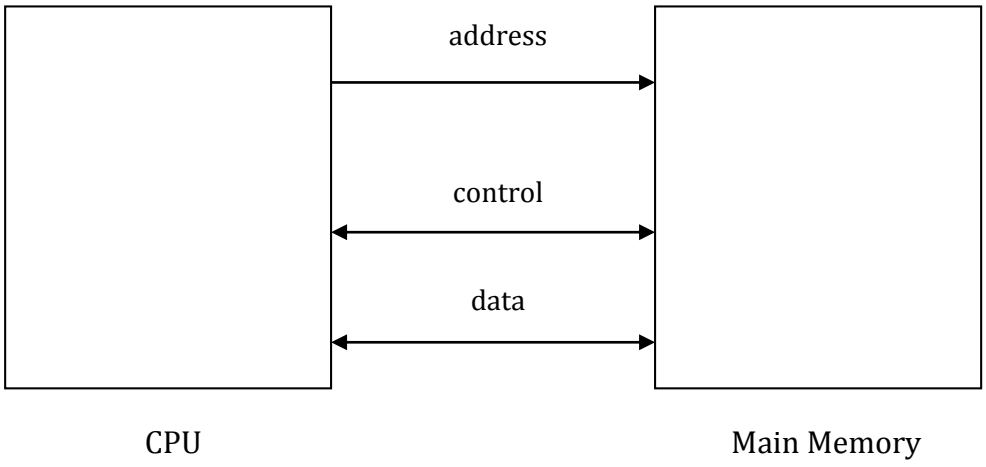
حل:

$$\text{زمان پاسخ} = 0.99 \times (10ns) + 0.01 \times (1000ns) = 19.9ns$$

در این مثال تأثیر وجود حافظه‌ی نهان در کاهش زمان دسترسی به روشنی دیده می‌شود.

حافظه‌ی نهان

بر اساس مدل فون نیومان نحوه‌ی ارتباط میان واحد پردازش و حافظه‌ی یک کامپیوتر به صورت زیر است:

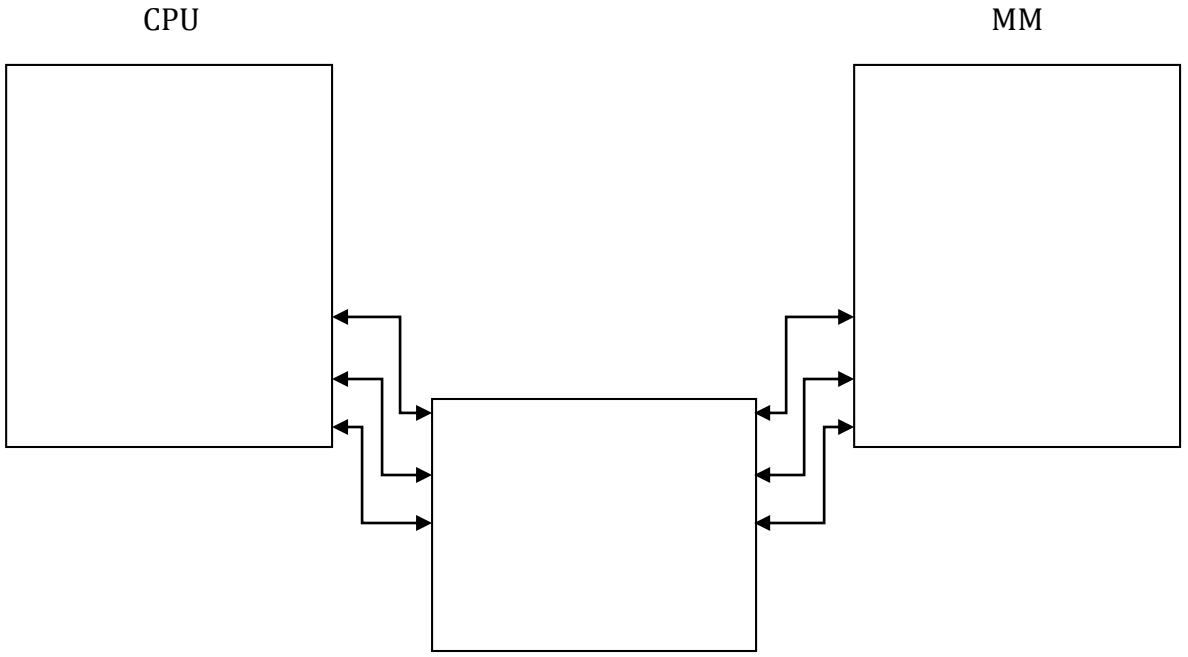


در طی سـالیان، این دو مؤلفه، یعنی Main Memory و CPU پیشـرفت کردند اما همواره فاصـله‌ای بزرگ میان این دو وجود داشـت. مثلاً CPU با سـیگنال سـاعت کار می‌کرد اما Main Memory اینگونه نبود. به این ترتیب سـیگنال سـاعت خیلی کوچک شـد و CPU رشد کرد اما Main Memory رشد چندانی نیافت. به تعبیری دیگر سرعت CPU زیاد شد در حالیکه Main Memory سرعت بالایی نداشت و این باعث می‌شد تا سرعت زیاد CPU چندان جلوه نکند. این مشکل هنوز هم در دنیای کامپیوتر حل نشده است.

ایده‌ی بهبود عملکرد: یک مسـئول کتابخانه را در نظر بگیرید. او از یک جعبه در کنار خود اسـتفاده می‌کند تا کتاب‌هایی را که بیشـتر مورد اسـتفاده قرار می‌گیرند در آن نگه دارد. معیار اینکه چه کتابی بیشتر مورد استفاده بوده، گذشته‌ی امانت کتاب‌هاست. به این ترتیب که هرگاه شما کتابی را که امانت گرفته‌اید به مسئول برگردانید،

او آن کتاب را در آن جعبه می‌گذارد. به عبارتی دیگر آن جعبه همواره شامل کتاب‌هایی است که اخیراً بیشتر استفاده شده‌اند. هر وقت هم که شما بخواهید کتابی را به امانت ببرید، ابتدا مسئول در جعبه به دنبال آن می‌گردد و در صورت یافت نشدن، آن را در مخزن پیدا می‌کند.

در مورد حافظه هم همین ایده را به کار بردند. به این ترتیب که بین CPU و Main Memory، بافری را قرار دادند تا کاری مشابه جعبه‌ی کتابدار داشته باشد، به این امید که این بافر بیشتر درخواست‌های CPU را hit کند.



این بافر در واقع همان حافظه‌ی نهان[۶] است(از آنجا که این بافر داده‌ها را در خود ذخیره می‌کند، نوعی حافظه است و از آنجا که بودن یا نبودن آن برای CPU تفاوتی ایجاد نمی‌کند و در واقع از دید CPU پنهان است، آن را حافظه‌ی نهان می‌نامند).

حال سؤال اساسی این است که در این حافظه چه اطلاعاتی را ذخیره کنیم بهتر است؟ اگر به این نکته توجه نکنیم، ممکن است بیشتر درخواست‌های CPU را miss کند و به این ترتیب زمان متوسط دسترسی به داده را بیشتر کند(چرا که اگر از همان اول بدانیم درخواست از حافظه‌ی نهان miss می‌شود، همه‌ی درخواست‌ها را مستقیماً از Main Memory می‌کردیم).

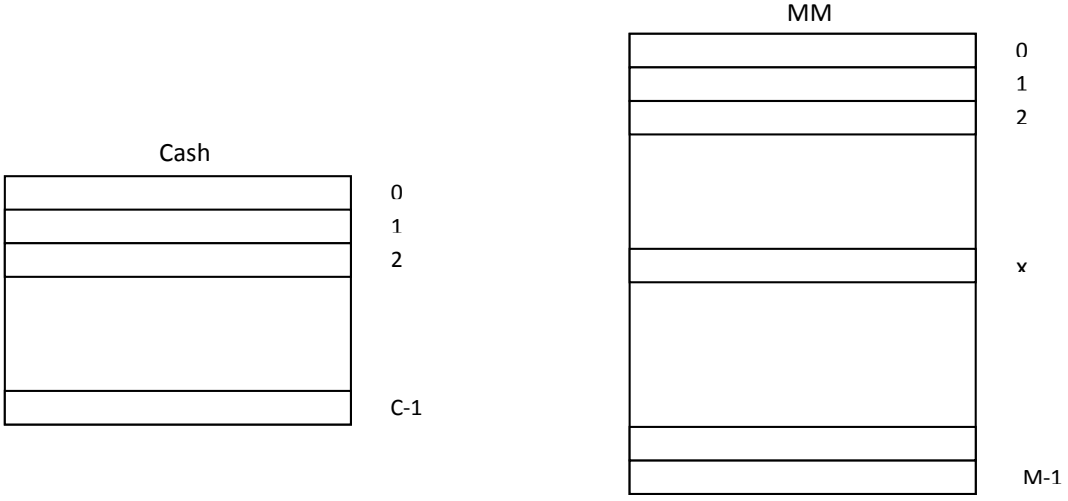
6 Cache

برای اینکه تصمیم بگیریم چه داده‌هایی را در حافظه‌ی نهان نگه‌داریم، براساس پیشینه‌ی درخواست‌ها از Main Memory عمـل می‌کنیم. برای این منظور انتقـالات داده بین CPU و Main Memory را با یک نظاره‌گر[7] روی هر سه Bus میانی بررسی کردند و متوجه شدند که درخواست‌ها سه خاصیت زیر را دارند:

- همجواری مکانی (Spatial locality): یعنی درخواســـت‌های متوالی از حافظه، از آدرس‌های نزدیک به هم در Main Memory اسـت و لذا درخواست‌های متوالی وابستگی مکانی دارند (مثلاً یک حلقه‌ی for را در نظر بگیرید که تعدادی دستورالعمل متوالیاً اجرا می‌شوند).
- همجواری زمـانی(Temporal Locality): یعنی یـک آدرس در زمـان‌های نزدیک به هم، چند بار مورد استفاده قرار می‌گیرد (مثلاً یک متغییر را در نظر بگیرید که چندین بار در طول اجرای یک برنامه از آن استفاده می‌شود).
- همجواری فرایندی(Process Locality): عملاً Taskهای مختلف ســیســـتم عامل و همچنین توابعی از برنامه‌ی درحال اجرا در حافظه نزدیک به هم ذخیره می‌شوند.

این ویژگی‌ها را در طراحی حافظه‌ی نهان منظور خواهیم کرد.

فرض کنید حافظه‌ی نهان C خط[8] داشـته باشـد که هر خط یک کلمه[9] است. متعاقباً فرض کنید حافظه M خط دارد.



7 Monitor
8 line
9 word (تعداد بایت‌های هر درخواست که 1، 2 یا 4 است)

در همین ابتدای کار باید بپرسیم داده‌های Main Memory را چگونه در حافظه‌ی نهان قرار دهیم و همینطور اگر بخواهیم داده‌ی جدیدی را از Main Memory بیاوریم، به جای چه داده‌ای از حافظه‌ی نهان باید جایگزین کنیم. به این ترتیب دوبحث اساسی در طراحی حافظه‌ی نهان مطرح می‌شود:

- سیاست جایدهی (Placement Policy)
- سیاست جایگزینی (Replacement Policy)

سیاست جایدهی و انواع حافظه‌ی نهان

الف) حافظه‌های نهان نگاشت مستقیم:

در وهله‌ی اول باید مکانیزمی ارائه کنیم تا خانه‌های Main Memory را به خانه‌های حافظه‌ی نهان نگاشت کند. به این مکانیزم Placement Mechanism یا Address Mapping می‌گویند. در مورد حافظه‌ی نهانی که پیش‌تر ارائه کردیم، ساده‌ترین مکانیزم برای نگاشت به صورت زیر است:

(آدرس در خافظه‌ی اصلی) Mod C = (آدرس در حافظه‌ی نهان)

در همین ساختار حافظه‌ی نهان فرض کنید داده‌ای از Main Memory با آدرس x را در حافظه‌ی نهان قرار داده‌ایم. حال اگر داده‌ی با آدرس x+C را از Main Memory بخواهیم، چون ابتدا به حافظه‌ی نهان درخواست می‌دهیم نمی‌توانیم بفهمیم داده‌ی ذخیره شده به آدرس x است یا x+C. به همین دلیل برای ذخیره‌ی هر داده در حافظه‌ی نهان، علاوه بر خود داده، باید مشخصاتی از آدرس داده در Main Memory را نیز ذخیره کنیم. اما آدرس در حافظه‌ی نهان، باقیمانده‌ی x به C است؛ لذا کافی است خارج‌قسمت تقسیم x به C را نگه داریم. به این عدد برچسب[10] می‌گویند.

اما هنوز این طراحی همجواری‌های مکانی را به خوبی لحاظ نکرده است. برای بهبود کارایی، به جای آنکه هر بار از Main Memory یک کلمه به حافظه‌ی نهان انتقال دهیم، یک بلوک B کلمه‌ای را انتقال می‌دهیم(البته روشن است که M، C و B همگی توان‌هایی از دو هستند و بنابراین مثلاً عمل تقسیم یک آدرس بر B جز یک شیفت دادن ساده نیست). بنابراین هم Main Memory و هم حافظه‌ی نهان را به بلوک‌های B کلمه‌ای تقسیم می‌کنیم و هرگاه بخواهیم داده‌ای را از Main Memory به حافظه‌ی نهان بیاوریم، کل بلوکی را که داده در آن است،

10 Tag

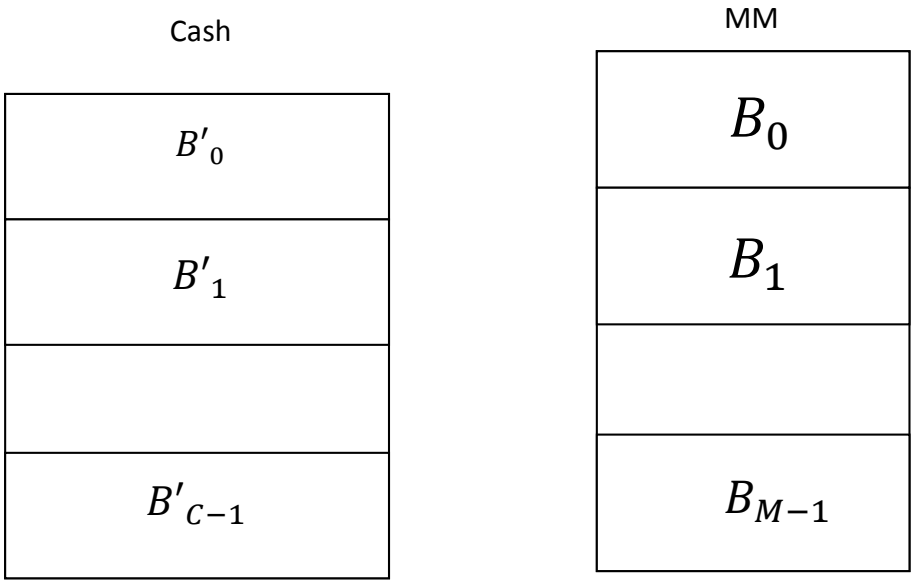
انتقال می‌دهیم. پس، از این به بعد فرض می‌کنیم Main Memory شـامل M بلوک B کلمه‌ای و حافظه‌ی نهان شامل C بلوک B کلمه‌ای است که:

$M = 2^m$

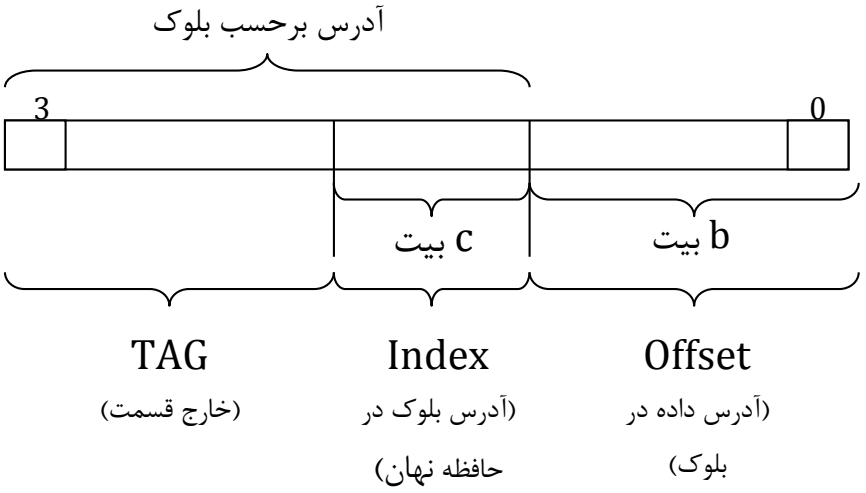
$C = 2^c$

$B = 2^b$

و لذا شِمای کلی به صورت زیر خواهد بود.



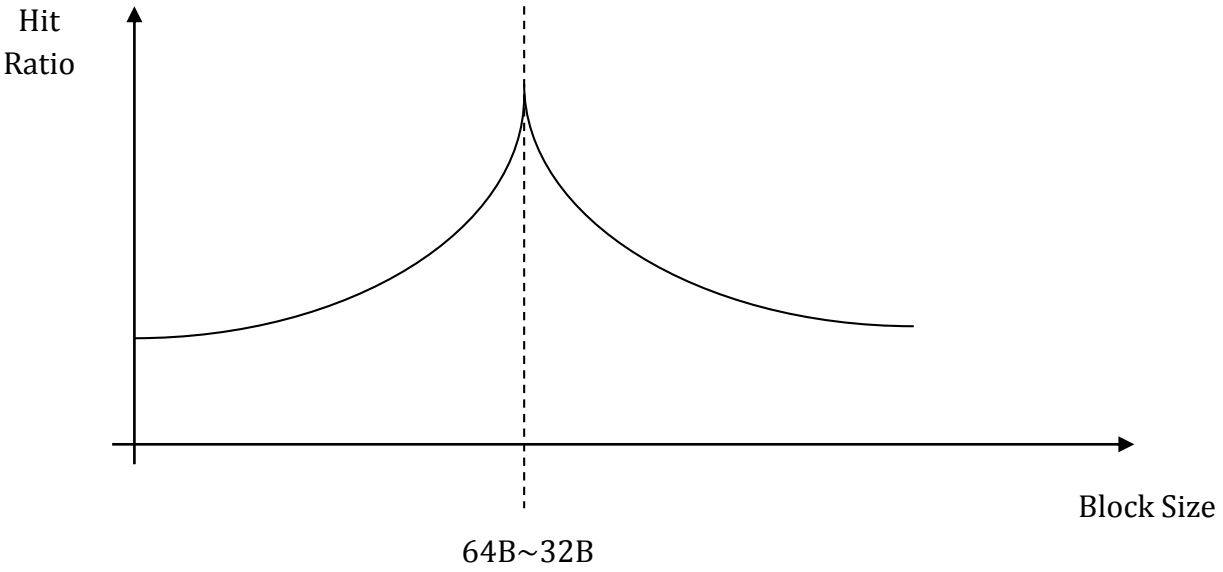
بدین ترتیب اگر آدرس‌ها مثلاً 32 بیتی باشند قالب آدرس CPU به صورت زیر است:



دقت کنید که تمام اعضای یک بلوک حافظه‌ی نهان برچسب یکسان دارند و لذا کافی است برای آنها، تنها یک عدد به عنوان برچسب نگه‌داری کنیم.

این طراحی حافظه‌ی نهان ساده‌ترین و کم‌خرج‌ترین نوع طراحی حافظه‌ی نهان است که به آن حافظه‌ی نهان با نگاشت مستقیم[11] می‌گویند که در عین سادگی Hit Ratio آن حدود 92 درصد است.

در مورد این نوع حافظه‌های نهان، در حجم ثابت می‌توان سایز بلوک را تغییر داد. با افزایش سایز بلوک، از یک سو همجواری‌های مکانی بیشتر خود را نشان می‌دهند و از سوی دیگر حافظه‌ی نهان را صلب کرده و تعداد کمتری بلوک را می‌توان نگه‌داری کرد. در بررسی‌های این نوع طراحی و تست کردن آن با برنامه‌های مختلف، نمودار Hit Ratio بر حسب سایز بلوک به صورت زیر بدست می‌آید:



تا اینجا بحث ما در مورد خواندن[12] داده از حافظه‌ی نهان و نحوه‌ی ارتباط آن با Main Memory بود. اما فرض کنید بخواهیم داده‌ای را بنویسیم یا به تعبیری دیگر بخواهیم محتویات خانه‌ای از حافظه را تغییر دهیم. در این صورت، درست مثل حالت خواندن ابتدا درخواستی به حافظه‌ی نهان می‌دهیم. دو حالت داریم:

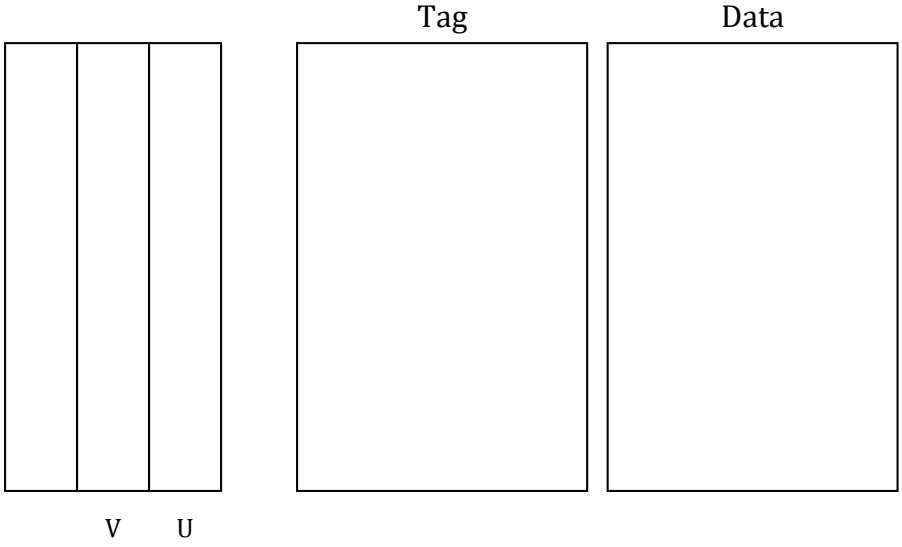
- Miss رخ دهد: در این صورت کافی است ابتدا داده در Main Memory پیدا شده، محتویاتش عوض شده و سپس به حافظه‌ی نهان انتقال یابد.

11 Direct Mapped Cache (DMC)
12 Read

- Hit رخ دهد: در این صـورت داده در حافظه‌ی نهان موجود اسـت و باید بروز شود و این بروزرسانی باید در Main Memory نیز صورت گیرد، اما چه وقت؟ برای این منظور دو مکانیزم وجود دارد:

☑ Write Through: بـه این معنی کـه هر وقـت داده‌ای در حافظه‌ی نهان بروز شـــد، محتویات آدرس متناظر در Main Memory هم بروز شود.

☑ Write Back: به این معنی که هر وقت خواستیم بلوکی را در حافظه‌ی نهان نابود کرده و بلوک دیگری را جـای آن قرار دهیم، در صـــورتی کـه محتویات داده‌های بلوک تغییر کرده بود، ابتدا داده‌هـای متناظر در Main Memory را بروز کرده و بعد بلوک جدید را جایگزین می‌کنیم. در این حالت برای اینکه بدانیم محتویات چه داده‌ای در حافظه‌ی نهان تغییر کرده است، می‌بایست یک بیت اضافه (u)[۱۳] برای هر بلوک نگه داریم.



البته روشـن اسـت که Write Back کارایی بهتری دارد اما در مواقعی که ممکن اسـت داده‌ها حافظه‌ی نهان از بین بروند، چندان مناسب نیست. [۱۴]

13 در عمل برای هر بلوک از حافظه‌ی نهان، چند بیت نگه‌داری می‌شود. ازجمله‌ی این بیت‌ها یکی u برای تشخیص بروز شدن داده‌های بلوک و یکی v برای تشخیص معتبر(valid) بودن داده‌های بلوک است. بیت v برای وقتی مناسب است که کامپیوتر تازه روشن شده و هنوز حافظه‌ی نهان خالی است.

14 براساس قانون مور، تعداد ترانزیستورها در واحد سطح هر چندوقت(حدود 18 ماه) یکبار دو برابر می‌شود. دو برابر شده تعداد به معنی کوچک شدن ابعاد ترانزیستورها و خازن‌هاست که طبق رابطه‌ی $C = \epsilon_0 \frac{A}{d}$، میزان بار خازن کمتر شده و لذا خازن نسبت به نویز حساس‌تر می‌شود. به این ترتیب کافی است ذره‌ای مانند ذره‌ی α به خازن برخورد کرده و آن را دشارژ کند که این به معنی از دست رفتن داده‌ی ذخیره شده در خازن (در واقع SRAM) است.

ب) حافظه‌های نهان انجمنی:

با اینکه در حافظه‌های نهان نگاشت مستقیم، Hit Ratio، حدود 92 درصد است، اما در بعضی مواقع کارایی خیلی بدی دارند. حالتی را در نظر بگیرید که آدرس‌های درخواست شده فقط مربوط به دو بلوک باشند و هر دوی این بلوک‌ها به یک جای حافظه‌ی نهان نگاشت شوند. به عنوان مثال فرض کنید حافظه‌ی نهانی با سایز 8 بایت داریم و همه‌ی درخواست‌ها از Main Memory بایت به بایت باشد. حال اگر دنباله‌ی درخواست‌ها از Main Memory به صورت 0 و 8 و 0 و 8 و .. باشد، همواره درخواست‌های از حافظه‌ی نهان به miss منجر خواهد شد. در واقع در این‌جا با اینکه 7 خانه از حافظه‌ی نهان، خالی و بی‌استفاده است، همواره از یک خانه‌ی آن استفاده می‌کنیم. برای رفع مشکل می‌توان به جای 8 خط یک بایتی، 4 خط دو بایتی در نظر گرفت:

0	0	8
1		
2		
3		

البته در این حالت نیز با دنباله‌ی 0, 4, 8, 12, 0, 4, 8, 12, 0, .. تمام درخواست‌ها miss خواهد شد.

به این مکانیزم حافظه‌ی نهان با مجموعه‌ی انجمنی[15] می‌گویند. در این نوع حافظه‌های نهان، به هر سطر یک مجموعه[16] گفته می‌شود. مسلماً تعداد بلوک‌های هر مجموعه مهم است و لذا برای معرفی این نوع از حافظه‌های نهان از واژه‌ی kWSA [17] (مثلاً در اینجا 2WSA) استفاده می‌کنند.

بدیهی است با افزایش k مقدار Hit Ratio و در نتیجه کارایی بیشتر می‌شود.
ماکزیمم مقدار k زمانی است که k برابر سایز حافظه‌ی نهان شود و عملاً فقط یک مجموعه داشته باشیم. به این نوع از حافظه‌های نهان، تمام انجمنی[18] می‌گویند.

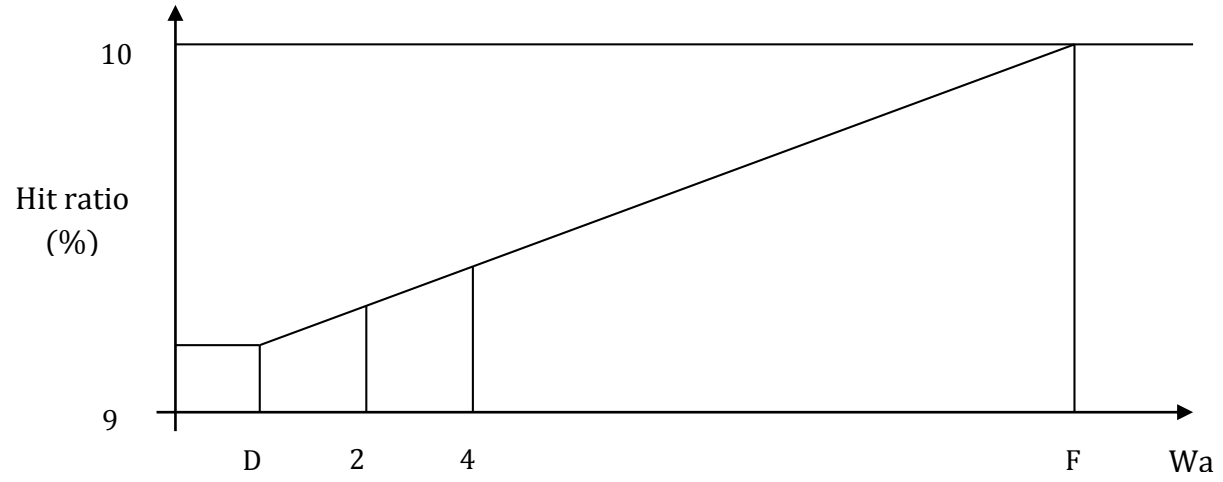
بگذارید موضوع را با یک مثال بهتر توضیح دهیم. فرض کنید یک کلاس با 50 صندلی داریم که استادی در این کلاس نشسته که با تعدادی از بچه‌های دانشگاه کار دارد که البته ممکن است با بعضی از بچه‌ها بیش از یک بار در زمان‌های مختلف کار داشته باشد. بالطبع اگر دانشجویی در کلاس نباشد و استاد هم با او کار داشته باشد باید کل دانشگاه جستجو شده تا آن دانشجو را پیدا کرده و به کلاس بیاوند؛ که این کار بسیار زمان‌گیر است. پس

15 Set Associative Cache (SAC)
16 Set
17 k-Way Set Associative
18 Fully Associative (FA)

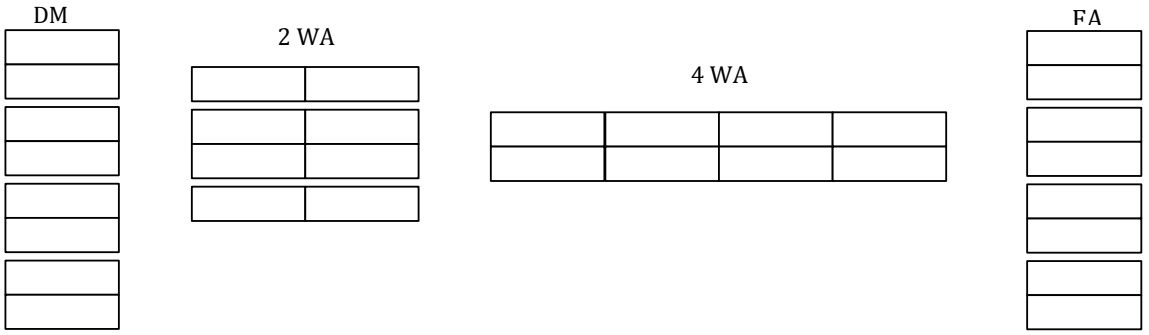
بهتر آن است که اگر دانشجویی به کلاس آمد، تا آنجاکه می‌توانیم او را نگه داریم. حال برای جایدهی دانشجویان در کلاس چند مکانیزم می‌توان اجرا کرد:

- صـــندلی‌ها را از 0 تا 49 شـــماره‌گذاری کنیم و برای هر دانشـــجویی که وارد کلاس می‌شـــود، ابتدا باقیمانده‌ی شـماره‌ی دانشـجویی او را به 50 محاسـبه کرده و سپس روی صندلی با آن شماره بنشانیم. اگر صندلی پر بود، دانشجوی قبلی را بیرون بیندازیم. این همان روشی است که DMC دارد.
- صـندلی‌ها را زوج زوج کنیم و این زوج‌ها را از 0 تا 25 شماره‌گذاری کنیم. حال برای هر دانشجویی که وارد کلاس می‌شـود، ابتدا باقیمانده‌ی شماره‌ی دانشجویی او را به 25 محاسبه کرده و او را روی یکی از صـندلی‌های زوج با آن شماره می‌نشانیم. اگر هر دو صندلی آن زوج، پر بودند، یکی از دانشجویان در آن زوج را بیرون می‌اندازیم. این همان روش 2WSA است.
- هر دانشـجوی جدیدی که وارد شـد، روی هر صـندلی که خالی بود بنشـیند و اگر صندلی خالی موجود نیست یکی از افراد قبلی را بیرون می‌اندازیم. این همان مکانیزم FA است.

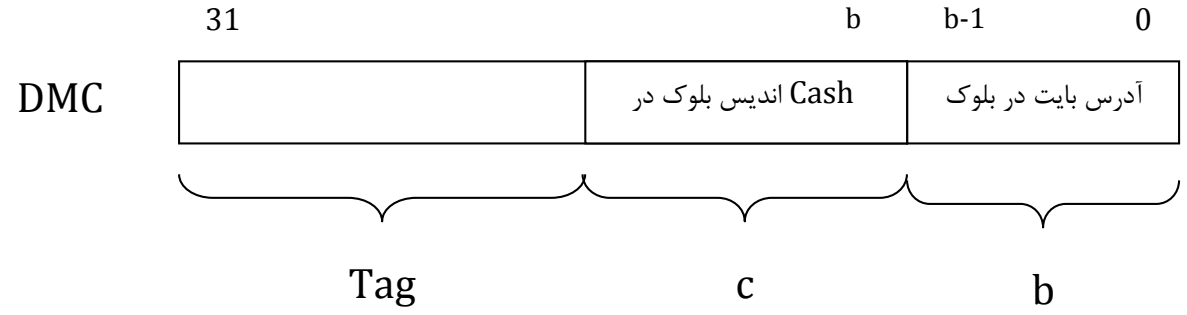
طبیعی اسـت اگر فرایند بیرون انداختن دانشجو در کلاس به صورت هوشمندانه‌ای باشد، راه‌حل سوم Hit Ratio بالاتری دارد. البته در راه‌حل سوم دسترسی به دانشجو زمان بیشتری می‌گیرد. در واقعیت داریم:

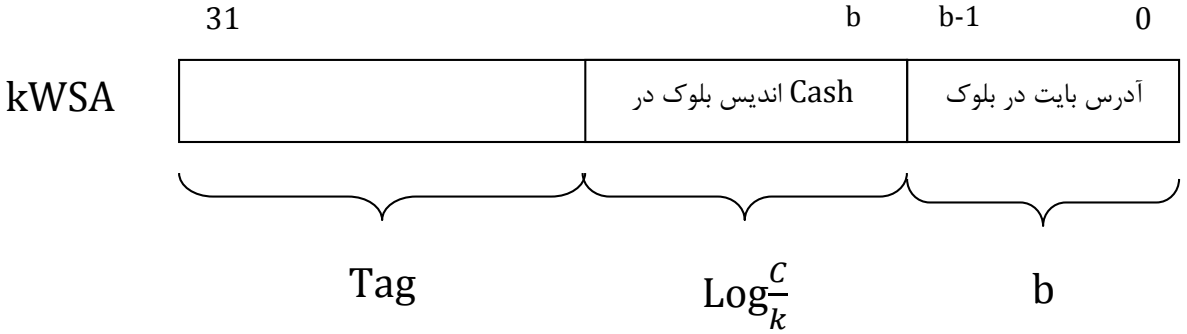


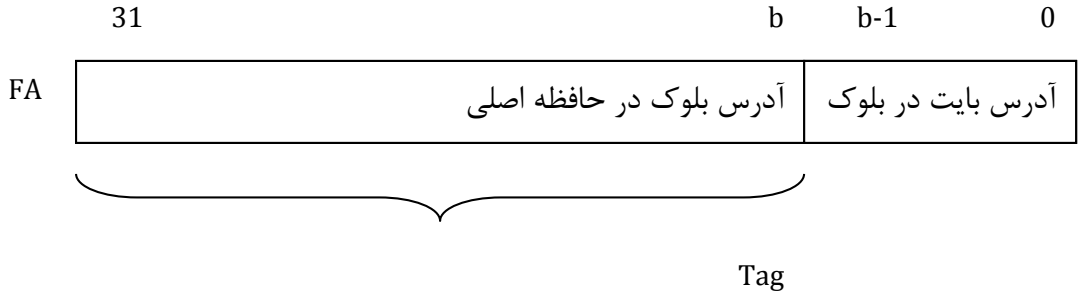
ذکر این نکته ضروری است که گاهی FA را همانند DM به صورت عمودی نشان می‌دهند. به عنوان مثال:



به طور خلاصه قالب آدرس در انواع مختلف حافظه‌ی نهان در زیر آمده است:



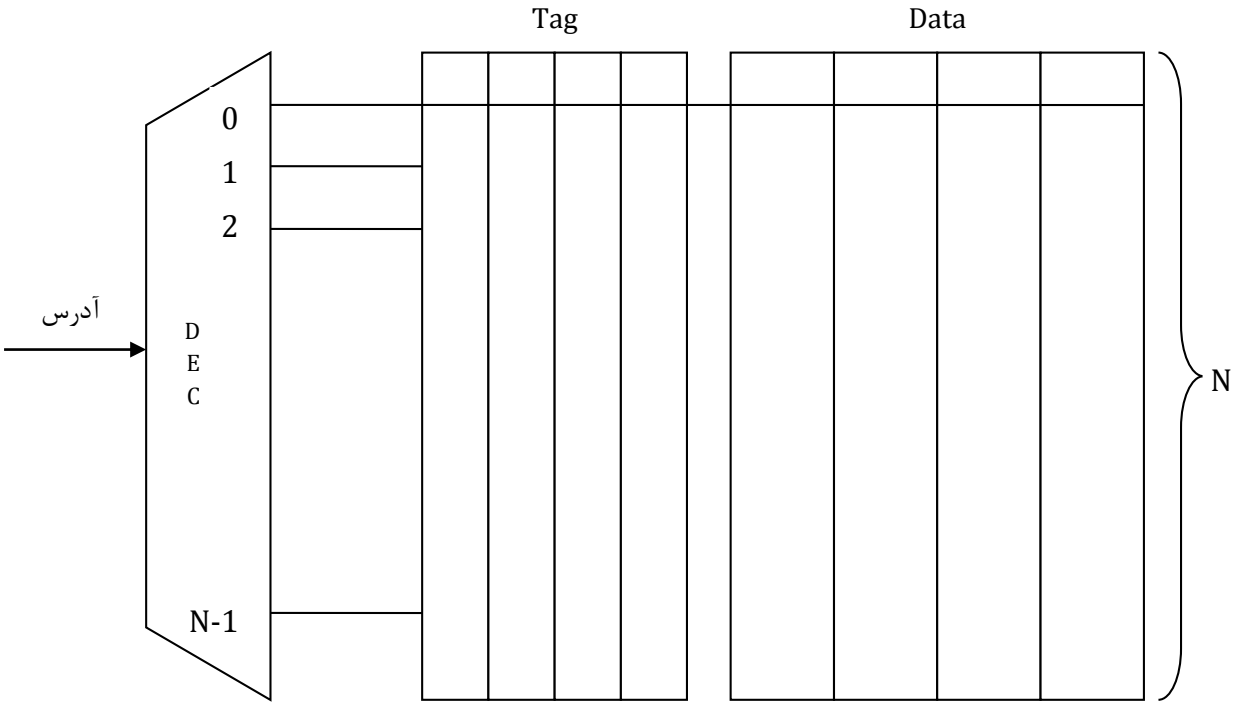




پ) طراحی مداری حافظه‌های نهان:

در حالت کلی طراحی kWSA را بررسی می‌کنیم. در این نوع حافظه‌های نهان بلوک‌های یک مجموعه برچسب یکسان ندارند. پس باید برای هرمجموعه k برچسب نگه‌داری کنیم:

وقتی آدرس از CPU وارد حافظه‌ی نهان شد، ابتدا اندیس مجموعه از آن جدا شده و به DEC داده می‌شود. پس از آن یک خط روشن شده ومقایرش به یک خط، حافظه‌ی، جدا (مانند M) منتقل می‌شود. برچسب آدرس ورودی با تمام k برچبب ذخیره شده در M مقایسه می‌شود (پس نیاز به مقایسه کننده داریم). از هر مقایسه عددی (در صورت تساوی صفر و گرنه غیر صفر) بدست می‌آید. پس k عدد خواهیم داشت که یا همه غیرصفرند (miss) و یا یک و فقط یکی از آنها صفر است (hit). در حالت hit باید مداری کنترلی، داده‌ی متناظر با برچسب مناسب را برگرداند.



اگر k=1 (DMC)، سایز DEC بزرگ شده اما دیگر نیازی به مقایسه‌کننده نیست و سایز برچسب حداقل است. در این حالت مدار کنترلی بسیار ساده است.

اگر FA باشد، نیازی به DEC نیست اما به تعداد بلوک‌ها نیاز به مقایسه‌کننده داریم و همچنین مدار کنترلی نسبتاً پیچیده است. به خاطر وجود تعداد زیاد مقایسه‌کننده و همچنین به‌خاطر پیچیده شدن مدار کنترلی و

بزرگ شـده سـایز برچسب، توان مصرفی و مساحت این نوع حافظه‌ی نهان به شدت زیاد بوده و مقرون به صرفه نیست.

سیاست جایگزینی

مسـلماً بهترین سـیاسـت‌ها، سـیاسـت‌هایی‌اند که المان‌هایی را اضافه و حذف کند که در آینده به نفعمان باشد. هدفمان در این بخش این اسـت تا بررسـی کنیم چه سـیاسـت‌هایی برای جایگزینی داده‌ها در حافظه‌ی نهان مناسـب‌ترند. برای مثال در یک 4WSA، اگر قرار باشـد عنصری جدید وارد حافظه‌ی نهان کنیم ولی مجموعه‌ای که می‌خواهیم داده را به آن اضـافه کنیم پر باشد، کدامیک از چهار عنصر مجموعه را بیرون بیندازیم بهتر است؟ روشـن اسـت که بحث سـیاست جایگزینی در مورد DMCها مطرح نیست، چرا که در این نوع حافظه‌ی نهان هر مجموعه، تک عضوی است و در صورت نیاز به جایگزینی، عنصری که باید بیرون انداخته شود معلوم است. اما هر چه مقدار k بیشـتر شـود این مکانیزم پیچیده‌تر و در عین حال مهم‌تر خواهد شـد. انواع روش‌هایی که برای جایگزینی می‌توان تصور کرد عبارتند از:

- Random: عنصری را به تصادف از مجموعه بیرون بیندازیم.
- FIFO[19]: هر عنصری که دیرتر وارد شده را بیرون بیندازیم.
- LIFO[20]: هر عنصری که زودتر وارد شده را بیرون بیندازیم.
- LRU[21]: عنصری که اخیراً کمتر استفاده شده بیرون برود.
- MRU[22]: عنصری که اخیراً بیشتر استفاده شده بیرون برود.
- LFU[23]: عنصری که تا به حال کمتر استفاده شده بیرون برود.
- MFU[24]: عنصری که تا به حال بیشتر استفاده شده بیرون برود.

19 First In First Out
20 Last In First Out
21 Least Recently Used
22 Most Recently Used
23 Least Frequently Used
24 Most Frequently Used

در روش LFU نیاز به شـمارنده داریم، اما پهنای بیتی این شمارنده باید نامحدود باشد و به همین خاطر استفاده از آن چندان عملی نیست. اما این روش بهترین است، چرا که مکانیزم جایگزینی بر اساس کل تاریخچه‌ی عناصر است

پس از LFU، LRU بهترین کارایی را دارد و برای پیاده‌ســازی آن فقط نیاز به نگه‌داری یک عدد(Rank) برای هر عنصر داریم که در واقع در هر بار دسترسی به آن، Rank آن عنصر یک واحد افزایش خواهد یافت. در اینجا تنها ملاک جایگاه نسبی میزان استفاده عنصر از زمانی که در داخل کش آمده‌اند می‌باشد.

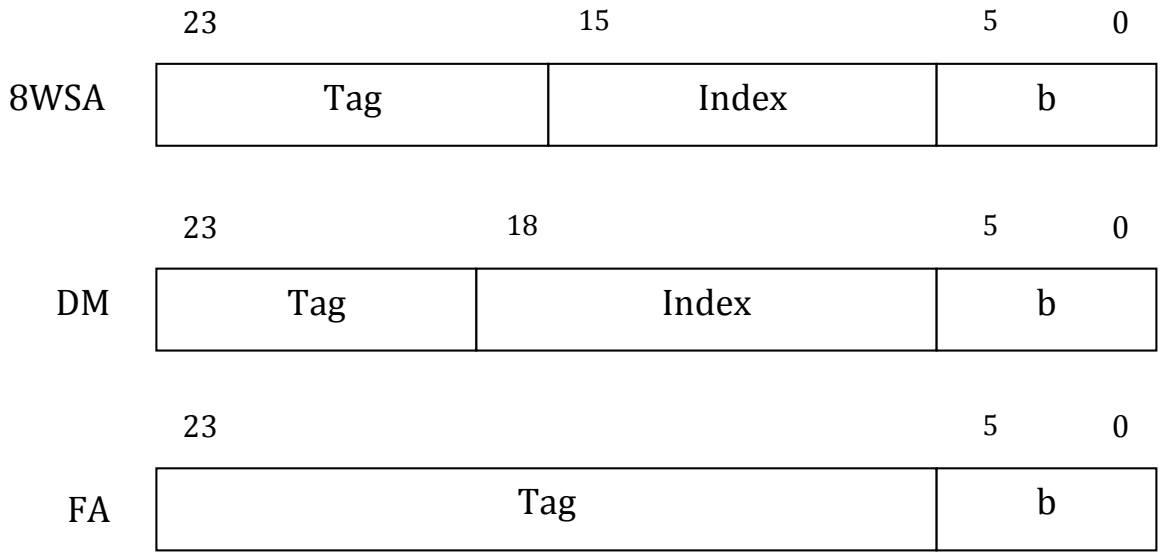
پس از LRU، Random کارایی نسـبتاً خوبی دارد و در حالتی که نمی‌خواهیم سـخت افزار اضافه‌تر برای Rank بگیریم مناسب است.

باقی مکانیزم‌ها چندان کارایی خوبی ندارند و در عمل استفاده نمی‌شوند.

مثال: چنانچه حافظه‌ی نهانی به اندازه‌ی 512KB و اندازه‌ی بلوک 64B داشــته باشــیم و اندازه‌ی حافظه‌ی اصـلی 16MB باشـد، مطلوب اسـت اندازه و قالب آدرس برای هر یک از حالات زیر:

8WSA
DM
FA

حل: حافظه‌ی اصلی 16MB و لذا آدرس‌ها 24bit هستند. بنابراین:



در حالت کلی: