دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

جزوه درس

معماری کامپیوتر

Computer Organization & Design

نسخه ۱,۵

گروه تدریسیاری

بهار ۱۳۹۶

فهرست

فصل اول

مروری بر مدار منطقی

و حافظه‌های رایانه

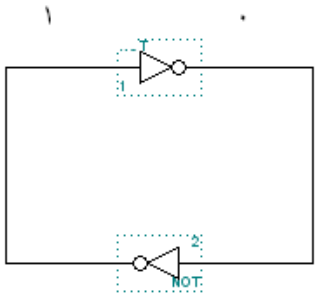
در این بخش در ابتدا یادآوری مختصری از درس مدار منطقی می‌شود سپس با زبان verilog آشنا می‌شویم که برای استفاده از آن برنامه ModelSim پیشنهاد می‌شود که در ضمیمه ۱ به طور جامعی مورد بررسی قرار گرفته است.

سپس با حافظه‌های رایانه آشنا می‌شویم. در ابتدا حافظه‌ها را بر اساس سرعت و هزینه در سلسله مراتب حافظـه مورد بررسی 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 اصلی و پردازشگر قرار می‌دهیم(cashe) و می‌کوشیم با شیوه‌های مختلف این ارتباط را سـریعتر کنیم

سپس فاکتورهای کارایی یک سیستم مورد بررسی قرار می‌گیرد.

Latch/Flip Flop

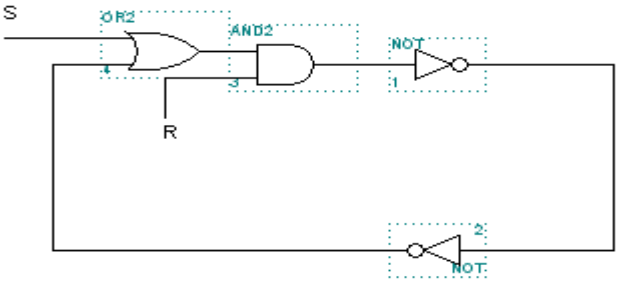
در مدار زیر می‌توانیم یک بیت اطلاعات ذخیره کنیم. اما هنگامی که مقدار داده شد دیگر نمی‌توان مقدار ذخیره شده را تغییر داد.



	R	S
نوشتن ۰	0	0
حافظه ای	1	0
نوشتن ۱	0	1
	1	1

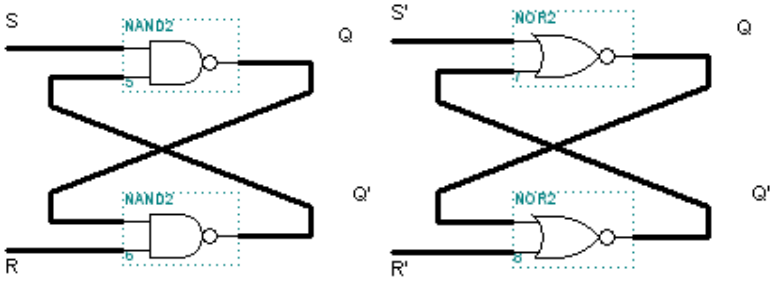
در مدار روبه رو می‌توان اطلاعات نیز ذخیره کرد، جدول صحت آن به شکل زیر است:

به این مدار Latch می‌گویند.



	R	S
تصادفی	0	0
نوشتن ۱	1	0
نوشتن ۰	0	1
حافظه ای	1	1

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

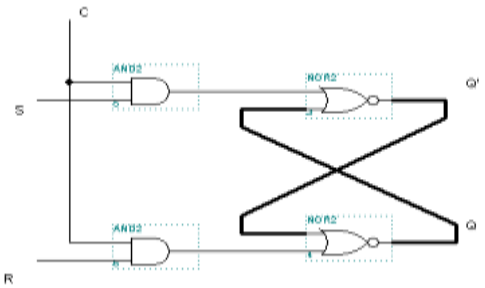


	R	S
تصادفی	0	0
نوشتن ۰	1	0
نوشتن ۱	0	1
حافظه ای	1	1

در هر دو حالت ۰-۰ 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.

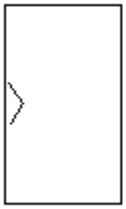


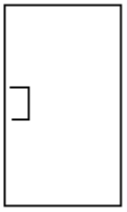
همانطور که مشهود است تنها زمانی مقدار نوشته می‌شود که $c = 1$ باشد.

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)

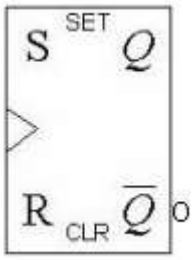




در شکل سمت راست فلیپ فلاپ حساس به سطح و در شکل سمت چپ فلیپ فلاپ حساس به لبه نمایش داده شده است.

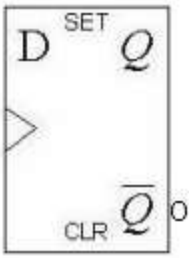
فلیپ فلاپ‌ها انواع مختلف دارند که به شرح زیر است:

RS-Flip Flop



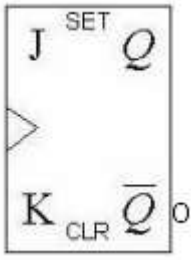
Inputs			Outputs		
S	R	C	Q	Q'	Comments
0	0	↑	Q	Q'	No change
0	1	↑	0	1	RESET
1	0	↑	1	0	SET
1	1	↑	?	?	Invalid

D-Flip Flop



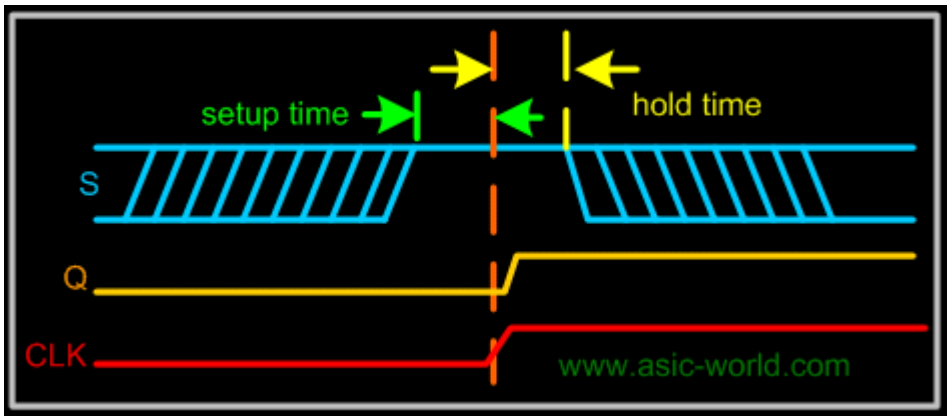
Inputs		Outputs		
D	C	Q	Q'	Comments
0	↑	0	1	RESET
1	↑	1	0	SET

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.

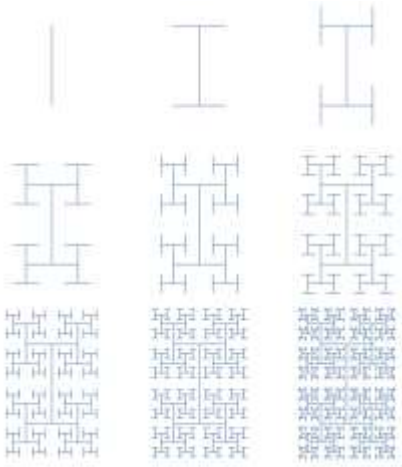
Hold time: کمترین بازه‌ی زمانی که مورد نیاز است ورودی بعد از گذار پالس ساعت پایدار باشد.

مقایسه‌ی مدارهای سنکرون و آسنکرون

مدارهای سنکرون	مدارهای آسنکرون
امکان قطع شدن مدار در آن وجود دارد. هم‌شنوایی[۱] رخ می‌دهد. (به خاطر حجم بالای سیم-ها) 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.

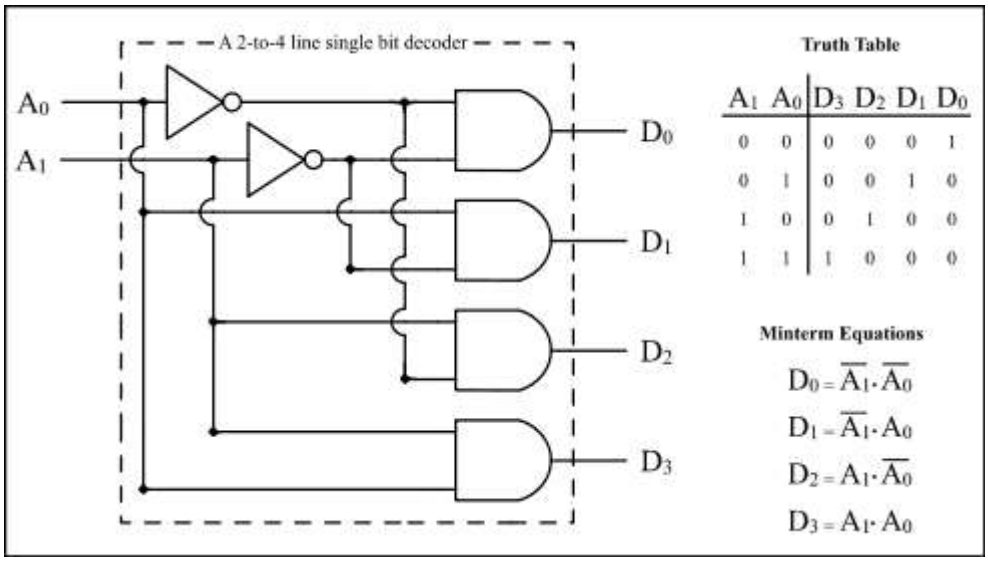
۱ CROSS TALK، یعنی مقدار سیم‌های همسایه روی هم تاثیر می‌گذارد.



شکل ۱ نمونه‌ای از H-Tree

Decoder

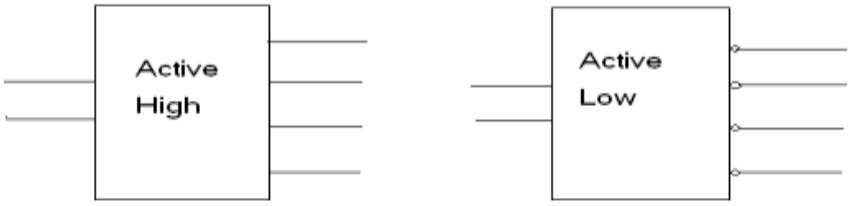
رمزگشا: این گونه عمل می‌کند که n خط ورودی دارد و بر حسب عدد ورودی یکی از 2^n خط خروجی(شماره ورودی) فعال شده و مابقی غیر فعال می‌شوند.



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

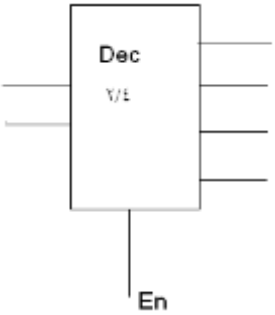
Active Low ⟶ یک خروجی صفر و باقی یک



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 مشخص می‌کند 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.

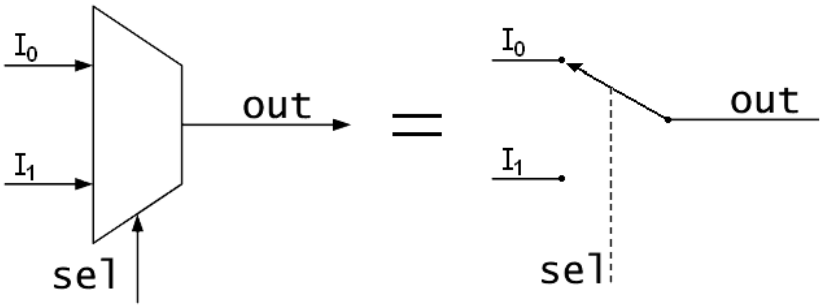
همانند Decoder دو منطق Active High و Active Low دارد.

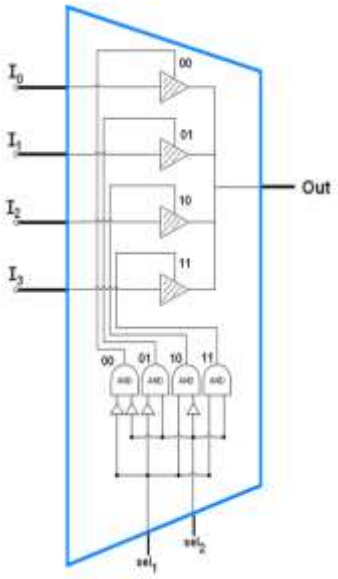
Encoder اولویت دار

برای ورودی‌ها هم اولویت قائل می‌شویم، به این ترتیب دیگر لزومی ندارد که در ورودی تنها یک خط فعال باشد. در خروجی Encoder اولویت دار چنانچه هیچ یک از ورودی‌ها فعال نباشند در خروجی خط z فعال می‌شود.

MUX

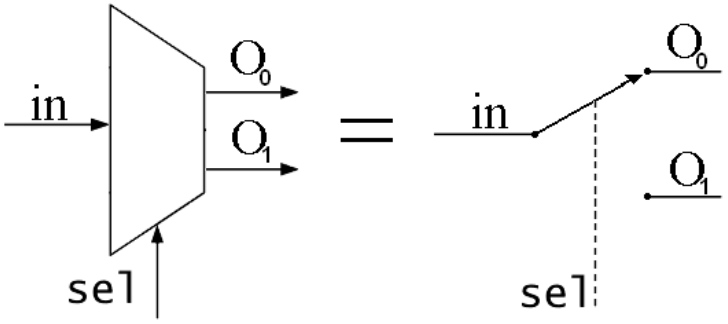
تسهیم کننده: 2^n خط ورودی و n خط انتخاب دارد و یک خط خروجی، بنا بر ورودی انتخاب خط ورودی را به خروجی منتقل می‌کند.

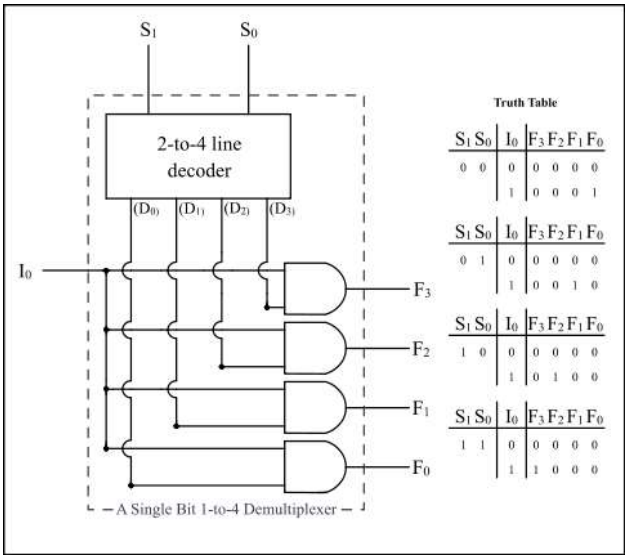




Demux

در حقیقت همان Decoder با ورودی Enable است.



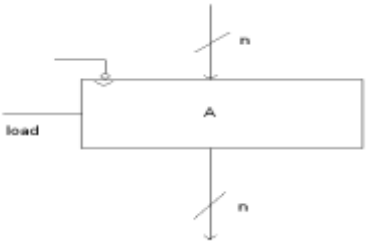


سوال: تفاوت Demux و Decoder در چیست؟

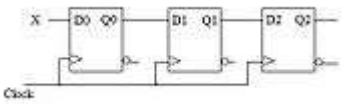
البته تفاوت‌های زیادی ممکن است به نظر برسد اما آنچه اینجا می‌خواهیم بگوییم این است که در Demux یک خروجی فعال و مابقی Z هستند اما در Decoder یک خروجی فعال و بقیه غیر فعال هستند.

Register

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



Parallel in-Parallel out



Shift Register

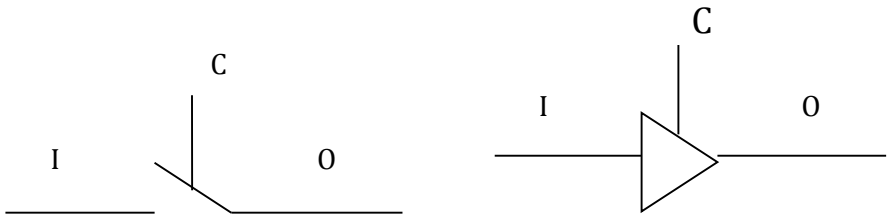
و از انواع دیگر می‌توان به Serial in-Serial out و Rotate اشاره کرد.

- 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 ۲ RAMها

داده‌های موجود در RAM قابل پاک شدن و جایگزینی با داده‌های دیگر هستند و هر نوع وقفه‌ای در جریان برق رایانه، موجب از بین رفتن داده‌های موجود در RAM می‌شود. البته در نوع خاصی از RAMها قابلیت نگهداری داده برای زمان طولانی‌تر وجود دارد. استفاده از این نوع حافظه‌ها، برای نگهداری موقت اطلاعات تا زمان پردازش یا انتقال نتایج به بیرون از رایانه و یا ذخیره در حافظه‌های جانبی است. داده‌های مورد نیاز پردازنده ابتدا وارد RAM شده و سپس پردازش روی آنها صورت می‌گیرد. به RAM، حافظه خواندنی و نوشتنی (RWM) هم میگویند.

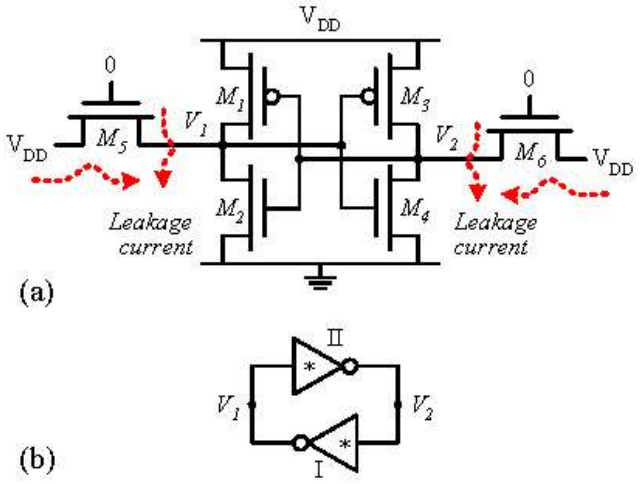
از نظر تکنولوژی ساخت، دو نوع RAM وجود دارد:

۱. (DRAM) Dynamic RAM

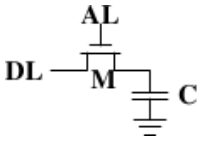
۲. (SRAM) Static RAM

DRAM نسبت به SRAM دارای سرعت دسترسی پایین‌تر و هزینه‌ی ساخت کمتر است. در این نوع حافظه اطلاعات باید به طور مرتب تجدید شوند وگرنه از بین خواهند رفت (البته این کار به صورت خودکار صورت می‌گیرد). از DRAMها در ساخت حافظه‌ی اصلی استفاده 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 ۳ ساختار داخلی SRAM



شکل ۴ ساختار داخلی DRA

در جدول زیر مقایسه‌ی این دو نوع RAM آمده است.

جدول ۱ 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 RAMاست که هم در لبه‌ی بالارونده و هم لبه‌ی پایین‌رونده Read, Write می‌کند.

ROM

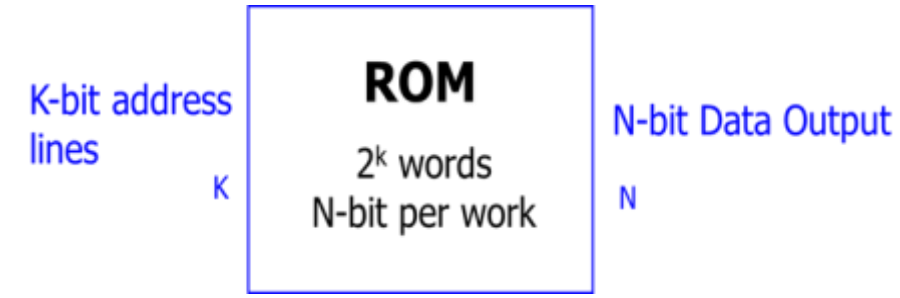
حافظه‌ای است فقط خواندنی که محتوی آن یکبار نوشته شده و پس از نصب در کامپیوتر تغییری در آن داده نمی‌شود.

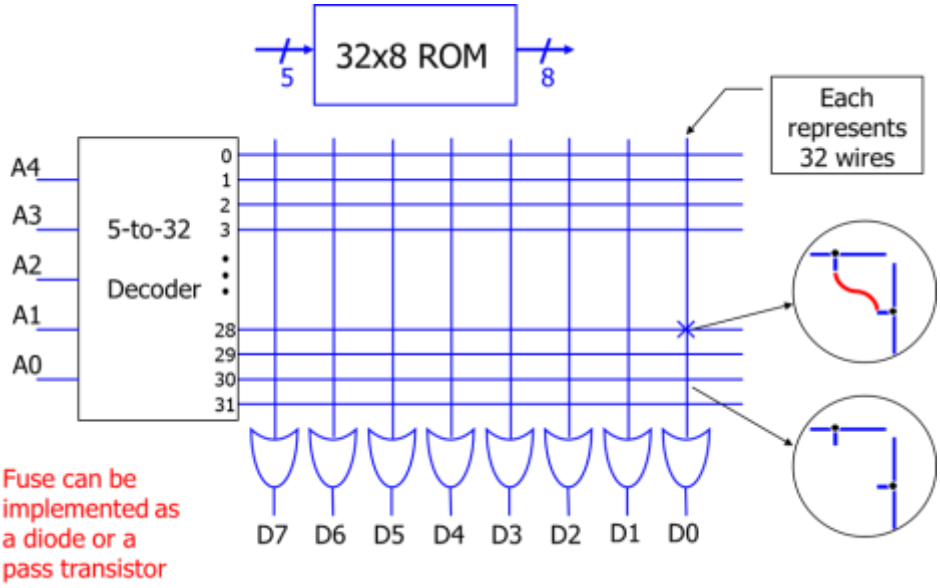
معمولا از این حافظه برای ذخیره برنامه هائی نظیر bootstrap loader که برای راه اندازی اولیه کامپیوتر مورد نیاز هستند استفاده می‌شود.

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.



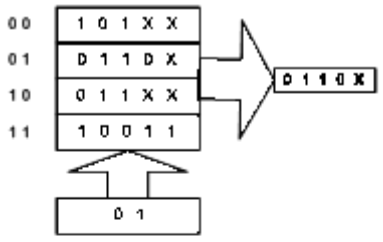


در این شکل یک ROM با ابعاد 8×32 آورده شده است که برای برنامه نویسی می‌بایست فیوز خط مربوطه را بسوزانیم.

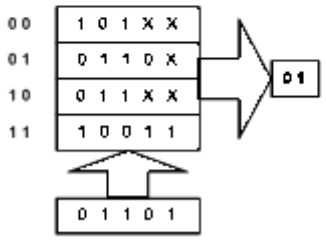
Content Addressable Memory (CAM)

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 ساخت حافظه‌هایی بوجود آمد که بتوانند بر اساس محتوا جستجو کنند.

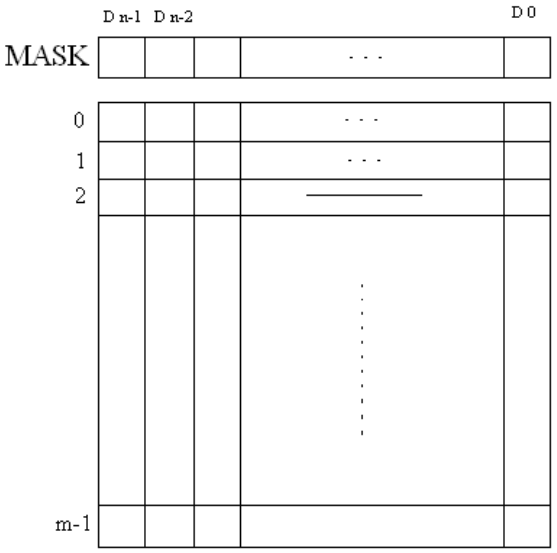


حافظه های آدرس پذیر



حافظه های **CAM**

همانطور که در شکل ملاحظه می‌کنید در حافظه‌های قدیمی با دادن آدرس، محتوای آدرس را دریافت می‌کردیم در حالیکه در حافظه‌ی CAM با دادن محتوا آدرس داده‌ی مشابه 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.



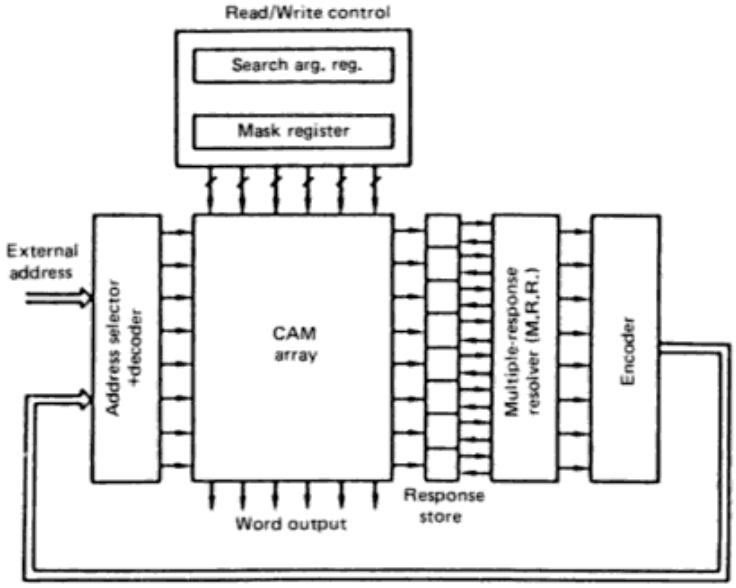
شکل ۵ حافظه‌ی CAM

شکل ۵ 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 می‌شود. به وضوح کلمه‌ای مورد نظر ماست 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 شد پس از XOR، برخی از کلمات کاملا با ورودی برابر می‌شوند 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 ۶ نمای کلی CAM در سیستم

۲ 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:

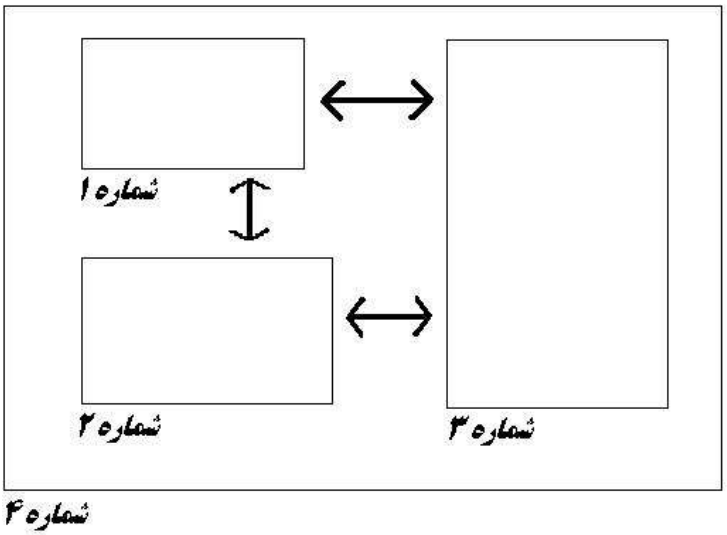
۱. Behavioral
   a. Initial statement
   b. Always statement
۲. Module instantiation
   a. برای شیئ گرفتن از ماژول‌های دیگر
۳. Gate instantiation
۴. UDP instantiation(User Defined Primitive)
   a. Gate Level
   b. Switch Level

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

۵. 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 ۴ تا ماژول باید تعریف کنیم. ابتدا ماژول‌های ۱ و ۲ و ۳ و در سپس شماره ۴ که باید در آن ۱ و ۲ و ۳ را instantiate کنیم.



چند نکته :

۱. UDP خاص کاربر است.

۲. در واقع خود UDPو گیت‌ها هم نوعی ماژول هستند.

۳. تمام مولفه‌ها به هم وصلند و با هم کار می‌کنند و هر مولفه به شرط ورودی گرفتن خروجی می‌دهد.

۴. مدارها باید پایدار باشند که وقتی ورودی گرفت بالاخره به یک حالت stable برسد.

معرفی کلمات کلیدی Verilog :

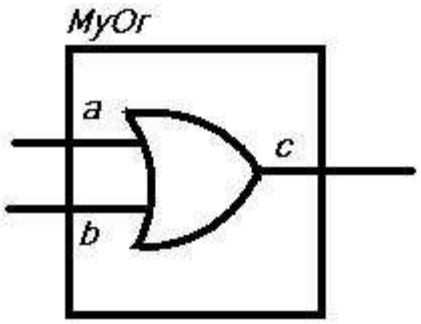
- reg یعنی رجیستر می‌خواهیم.
- Wire یعنی سیم می‌خواهم.

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.

۱. سطح گیت :

اسم ماژول هم اهمییتی ندارد و بیشتر برای سیمولاتور است.

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

۲. سطح RTL :

این همان مفهوم لحیم کاری در سخت افزار است.

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

عبارت بالا یک Continues Assignment است.

۳. سطح Behavioral:

Always statement همیشه 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 نیست چون سیمولاتور را مشغول 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 صورت ترتیبی اجرا می‌شوند. دستورات داخل این بلوک نیاز به assign ندارند و عملا دستورات نرم افزاری اند. توجه کنید که متغییرهایی که در سمت 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 هـم 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
```

Directiveهای مختلفی وجود دارد. مثلا undefined var یعنی چیزهایی که تعریف نکردیم را مثلا از نوع wire تعریف کنیم و یا:

```
‘Include A. v
```

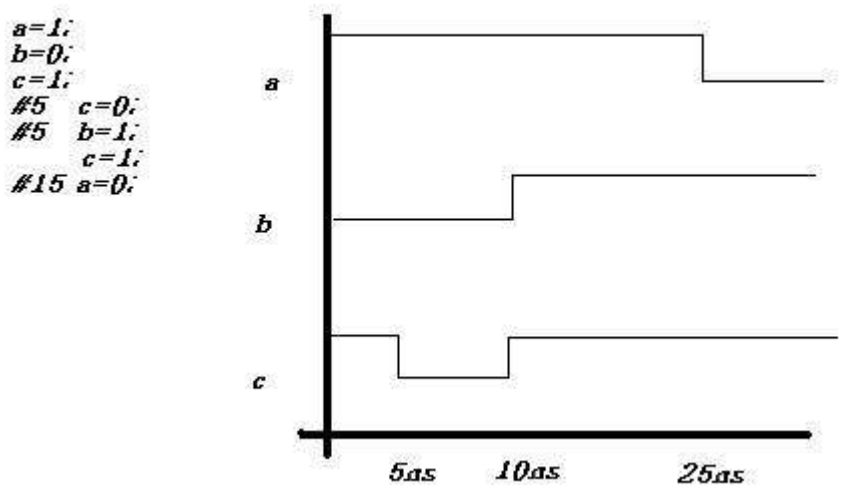
گاهی برای تاخیر میخواهیم نایستد. یعنی ;c=0 #5 باشد اما دستورات بعدی را همان لحظه ادامه دهد.

دراینجا دو نوع assignment داریم:

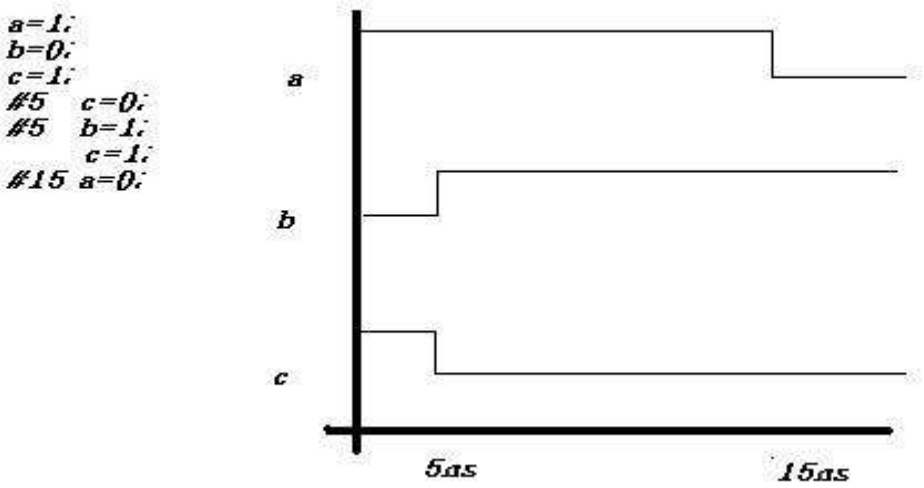
1. Blocking assignment
2. Non-blocking assignment

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 ۵و ۱۰و۲۵ 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 ۱ 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 ۱۰۲۴*۸ را انگونه میگوییم:

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

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

۷	۶	۵	۴	۳	۲	۱	۰
۰							
۰							
۰	ردیف ۱۰۲۳ام						

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 ۷ 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 شرکت سخت‌افزاری ساخت قطعات کامپیوتری هستیم و می‌خواهیم محصولی را ارزه کنیم که هم از نظر کارایی خوب باشد و هم قیمت بالایی نداشته باشد. برای دستیابی 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:

۱. Flip Flop
۲. Register
۳. 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 ۸ و حداکثر ۲۵۶تا) را در یک دسته قرار دهیم. هر کدام از این دسته‌ها را یک Register File\Bank می‌گویند.
۴. Cache
۵. Main Memory
۶. Magnetic Disk: مثل هارد و فلاپی و. ..
۷. Optical Disk: مثل CDها
۸. Tapeها و کارت پانچ‌ها: Tapeها برای ذخیره‌ی انبوهی از اطلاعات و به عنوان آرشیو مورد استفاده قرار می‌گیرند. تنها ضعف آنها ترتیبی[۳] 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:

---

۳ 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:

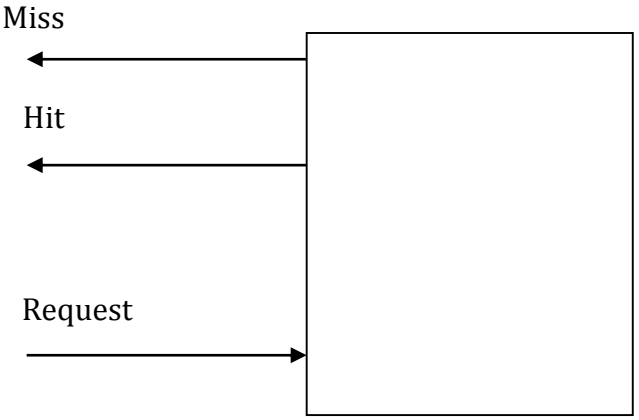
$d_1 \cong 4ns$

$d_4 \cong 10ns$

$d_6 \cong ms$

$d_8 \cong s$

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:

۴ 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[۵] بیش از ۹۴ 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 $d_i \gg d_{i-1}$ و لذا d_{i-1} نسبت به d_i قابل صرف نظر کردن است.

۵ Cache

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

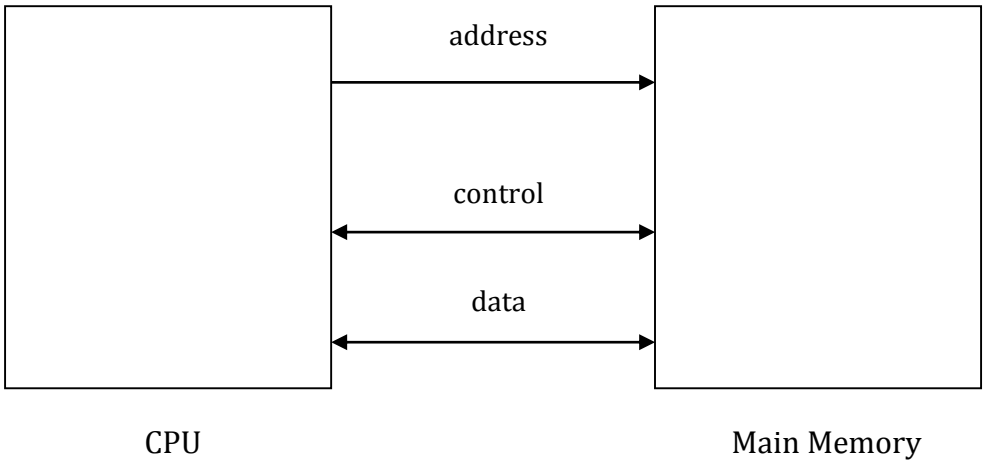
حل:

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

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:

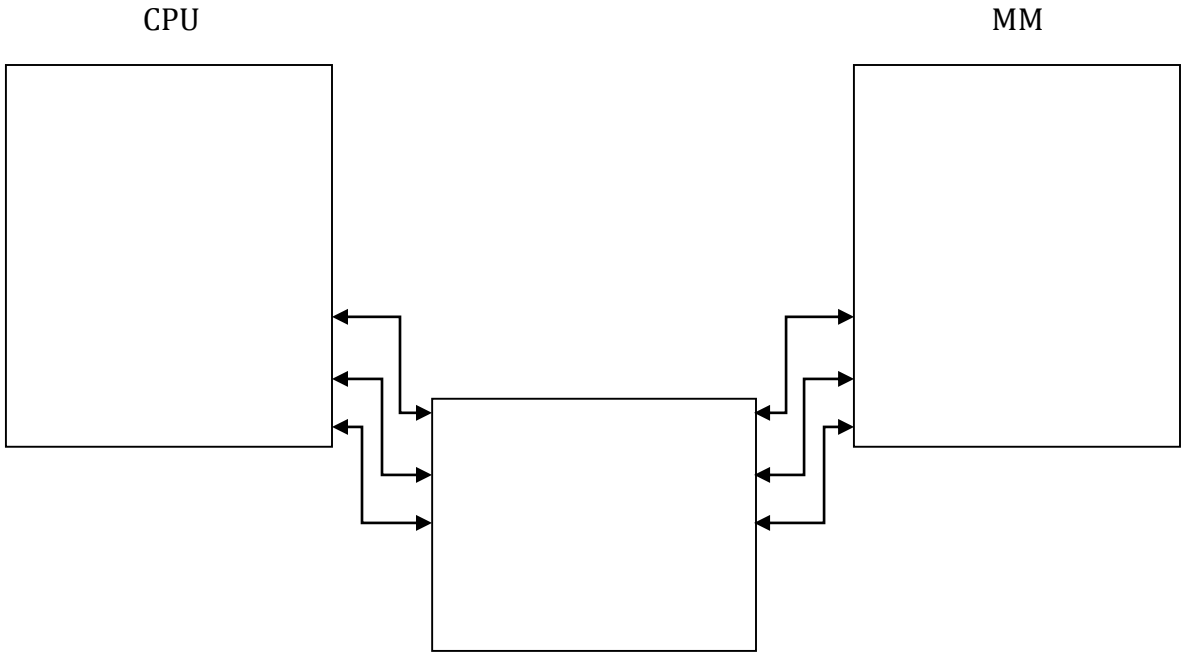


در طی سالیان، این دو مؤلفه، یعنی Main Memory و CPU پیشرفت کردند اما همواره فاصله‌ای بزرگ میان این دو وجود داشت. مثلاً CPU با سیگنال ساعت کار می‌کرد اما Main Memory اینگونه نبود. به این ترتیب سیگنال ساعت خیلی کوچک شد و CPU رشد کرد اما Main Memory رشد چندانی نیافت. به تعبیری دیگر سرعت CPU زیاد شد در حالیکه 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 مسئول کتابخانه را در نظر بگیرید. او از یک جعبه در کنار خود استفاده می‌کند تا کتاب‌هایی را که بیشتر مورد استفاده 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).

۶ Cache

برای اینکه تصمیم بگیریم چه داده‌هایی را در حافظه‌ی نهان نگه‌داریم، براساس پیشینه‌ی درخواست‌ها از Main Memory عمل می‌کنیم. برای این منظور انتقالات داده بین CPU و Main Memory را با یک نظاره‌گر[۷] روی هر سه Bus میانی بررسی کردند و متوجه شدند که درخواست‌ها سه خاصیت زیر را دارند:

- همجواری مکانی (Spatial locality): یعنی درخواست‌های متوالی از حافظه، از آدرس‌های نزدیک به هم در Main Memory است و لذا درخواست‌های متوالی وابستگی مکانی دارند (مثلاً یک حلقه‌ی for را در نظر بگیرید که تعدادی دستورالعمل متوالیاً اجرا می‌شوند).
- همجواری زمانی(Temporal Locality): یعنی یک آدرس در زمان‌های 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 سیستم 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.

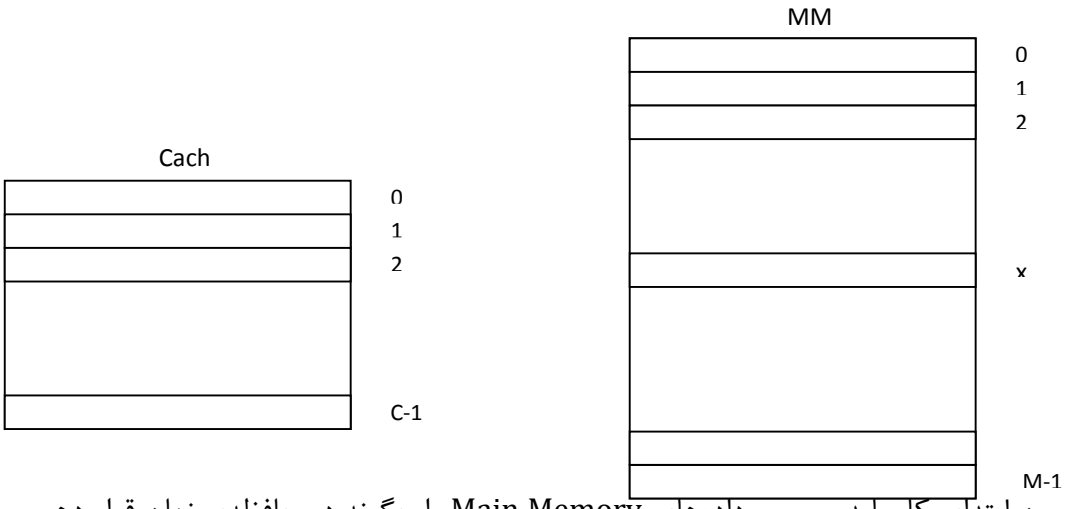
فرض کنید حافظه‌ی نهان C خط[۸] داشته باشد که هر خط یک کلمه[۹] است. متعاقباً فرض کنید حافظه M خط دارد.

---

۷ Monitor

۸ line

۹ word (تعداد بایت‌های هر درخواست که ۱، ۲ یا ۴ 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 حافظه‌ی 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 نگـه داریـم. بـه این عدد برچسب[۱۰] می‌گویند.

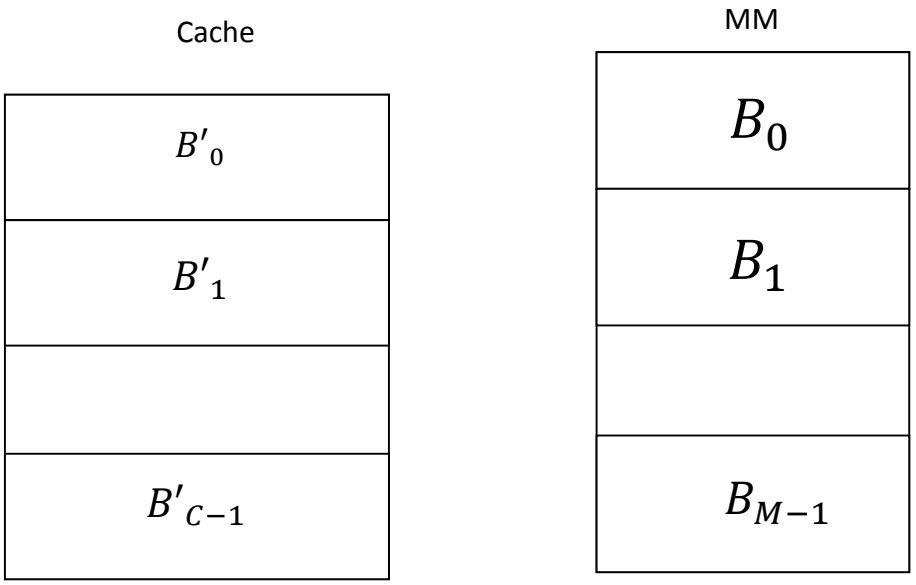
اما هنوز این طراحی همجواری‌های مکانی را به خوبی لحاظ نکرده است. برای بهبود کارایی، به جای آنکه هر بار از Main Memory یک کلمه به حافظه‌ی نهان انتقال دهیم، یـک بلـوک B کلمـه‌ای را انتقـال می‌دهیم(البتـه روشن است که M، C و B همگی توان‌هایی از دو هستند و بنابراین مثلاً عمل تقسیم یک آدرس بر B جـز یـک شیفت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

$M = 2^m$

$C = 2^c$

$B = 2^b$

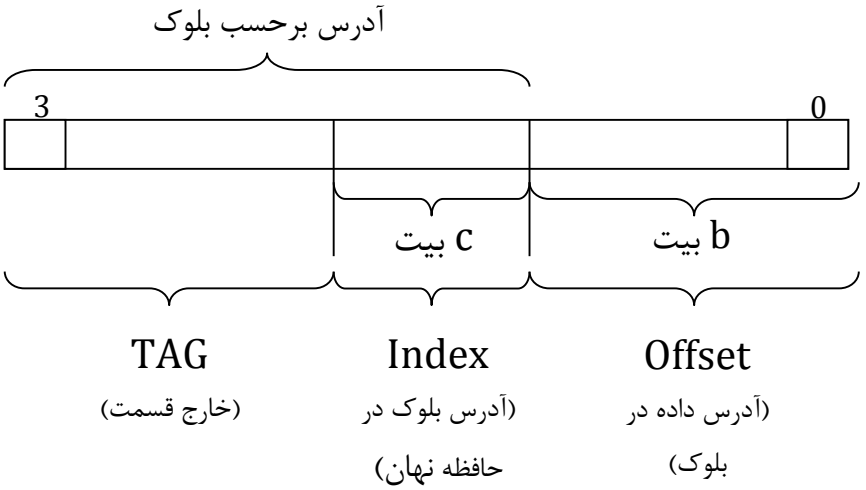
و لذا شِمای کلی به صورت زیر خواهد بود.



بدین ترتیب اگر آدرس‌ها مثلاً ۳۲ بیتی باشند قالب آدرس CPU به صورت زیر است:

---

۱۰ Tag

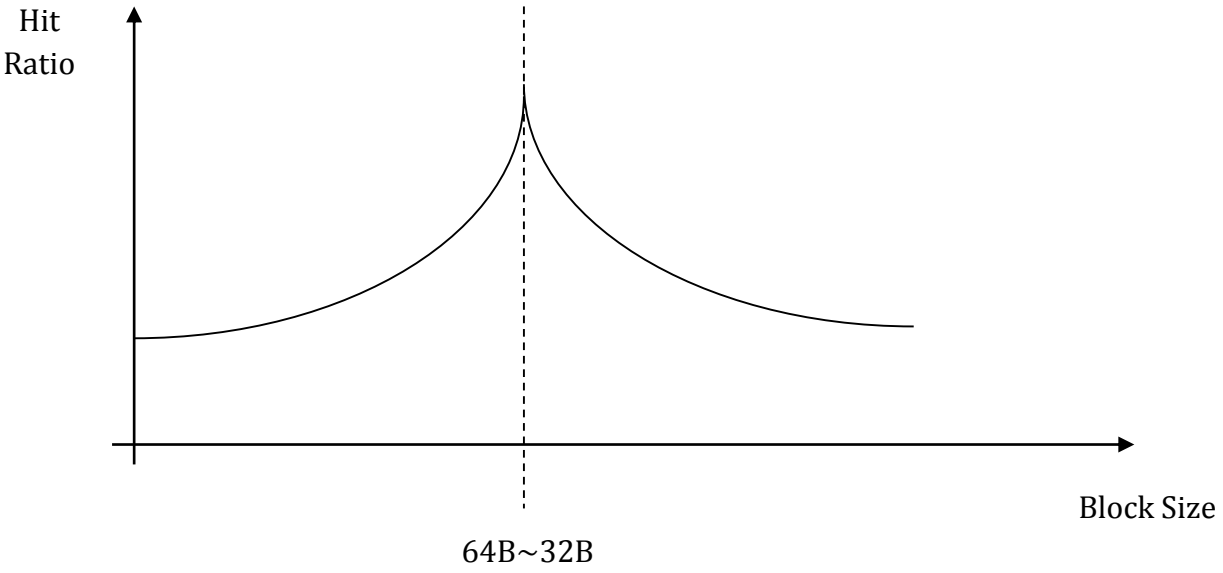


دقت کنید که تمام اعضای یک بلوک حافظه‌ی نهان برچسب یکسان دارند و لذا کافی است برای آنها، تنها یک عدد به عنوان برچسب نگه‌داری کنیم.

این طراحی حافظه‌ی نهان ساده‌ترین و کم‌خرج‌ترین نوع طراحی حافظه‌ی نهان است که به آن حافظه‌ی نهان با نگاشت مستقیم[۱۱] می‌گویند که در عین سادگی Hit Ratio آن حدود ۹۲ 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:

۱۱ Direct Mapped Cache (DMC)



تا اینجا بحث ما در مورد خواندن[۱۲] 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:

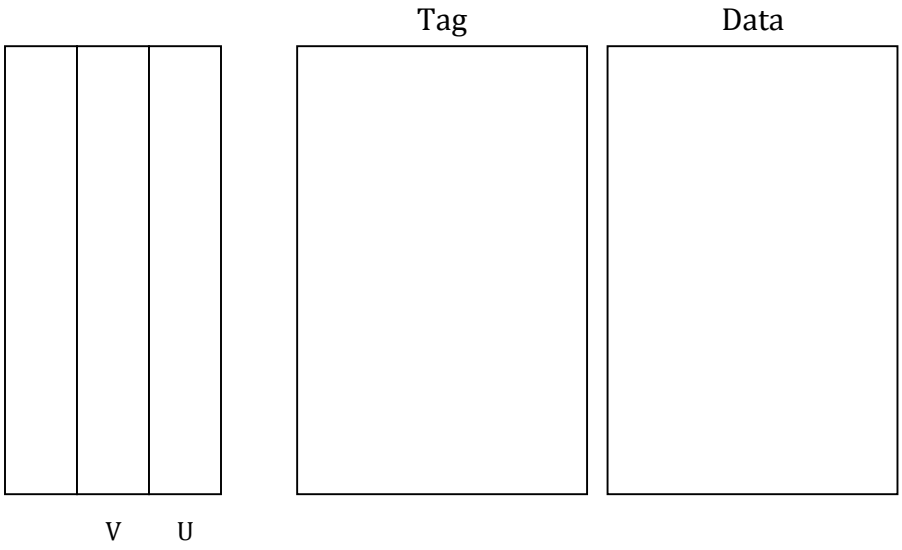
- Miss رخ دهد: در این صورت کافی است ابتدا داده در Main Memory پیدا شده، محتویاتش عوض شده و سپس به حافظه‌ی نهان انتقال یابد.
- Hit رخ دهد: در این صورت داده در حافظه‌ی نهان موجود است و باید بروز شود و این بروزرسانی باید در Main Memory نیز صورت گیرد، اما چه وقت؟ برای این منظور دو مکانیزم وجود دارد:

☑ Write Through: به این معنی که هر وقت داده‌ای در حافظه‌ی نهان بروز شد، 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 جای آن قرار دهیم، در صورتی که محتویات داده‌های بلوک تغییر کرده بود، ابتدا داده‌های متناظر در Main Memory را بروز کرده و بعد بلوک جدید را جایگزین می‌کنیم.

۱۲ Read

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

ب) حافظه‌های نهان انجمنی:

با اینکه در حافظه‌های نهان نگاشت مستقیم، Hit Ratio، حدود ۹۲ 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 ۸ بایت داریم و همه‌ی درخواست‌ها از Main Memory بایت به بایت باشد. حال اگر دنباله‌ی درخواست‌ها از Main Memory به صورت ۰ و ۸ و ۰ و ۸ و.. باشد، همواره درخواست‌های از حافظه‌ی نهان به miss منجر خواهد شد. در واقع در این‌جا با اینکه ۷ خانه از حافظه‌ی نهان، خالی و بی‌استفاده است، همواره از یک خانه‌ی آن استفاده می‌کنیم. برای رفع مشکل می‌توان به جای ۸ خط یک بایتی، ۴ خط دو بایتی در نظر گرفت:

---

۱۳ 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.

۱۴ براساس قانون مور، تعداد ترانزیستورها در واحد سطح هر چندوقت(حدود ۱۸ 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.

0	0	8
1		
2		
3		

البتـه در ایـن حالـت نیـز بـا دنبالـه‌ی 0, 4, 8, 12, 0, 4, 8, 12, 0, .. تمـام درخواست‌ها miss خواهد شد.

به این مکانیزم حافظه‌ی نهان با مجموعه‌ی انجمنی[۱۵] می‌گویند. در این نـوع حافظه‌های نهان، به هر سطر یک مجموعه[۱۶] گفتـه می‌شـود. مسـلماً تعـداد بلوک‌های هر مجموعه مهم است و لذا برای معرفی ایـن نـوع از حافظـه‌های نهان از واژه‌ی kWSA [۱۷] (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. بـه ایـن نـوع از حافظـه‌های نهـان، تمـام انجمنی[۱۸] می‌گویند.

بگذارید موضوع را با یک مثال بهتر توضیح دهیم. فرض کنید یک کلاس با ۵۰ 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 بـار در زمان‌های مختلف کار داشته باشد. بالطبع اگر دانشجویی در کلاس نباشد و استاد هم با او کـار داشـته باشـد باید کل دانشگاه جستجو شده تا آن دانشجو را پیدا کرده و به کلاس بیاوند؛ که این کار بسـیار زمـان‌گیر اسـت. پس بهتر آن است که اگر دانشجویی به کلاس آمد، تا آنجاکه می‌تـوانیم او را نگـه داریـم. حـال بـرای جایـدهی دانشجویان در کلاس چند مکانیزم می‌توان اجرا کرد:

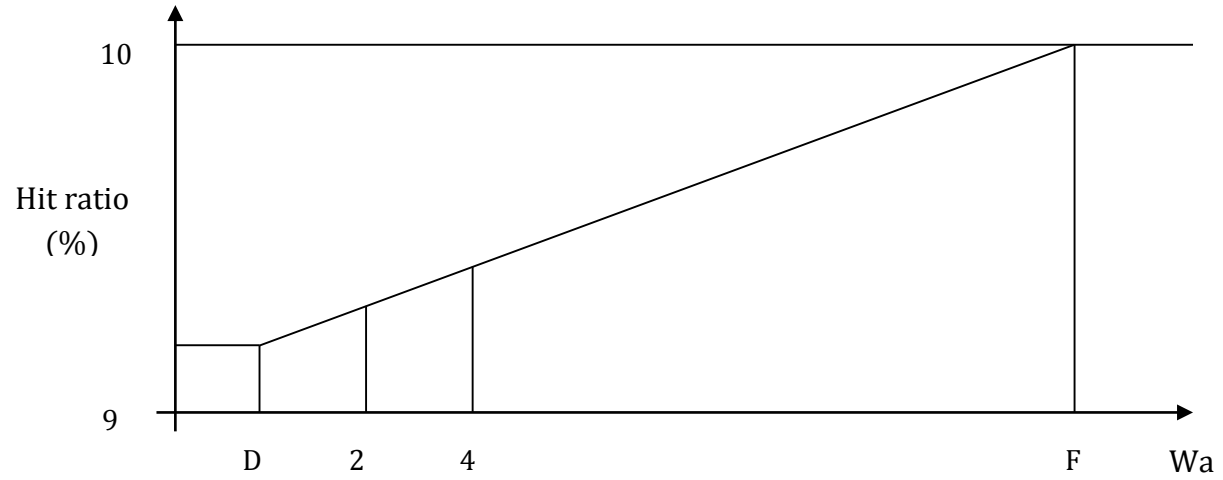
- صندلی‌ها را از ۰ تـا ۴۹ شـماره‌گذاری کنـیم و بـرای هـر دانشـجویی کـه وارد کـلاس می‌شـود، ابتـدا باقیمانده‌ی شماره‌ی دانشجویی او را به ۵۰ محاسبه کرده و سپس روی صندلی بـا آن شـماره بنشـانیم. اگر صندلی پر بود، دانشجوی قبلی را بیرون بیندازیم. این همان روشی است که DMC دارد.
- صندلی‌ها را زوج زوج کنیم و این زوج‌ها را از ۰ تا ۲۵ شماره‌گذاری کنیم. حال برای هر دانشجویی کـه وارد کلاس می‌شود، ابتدا باقیمانده‌ی شماره‌ی دانشجویی او را به ۲۵ محاسبه کرده و او را روی یکـی از

۱۵ Set Associative Cache (SAC)
۱۶ Set
۱۷ k-Way Set Associative
۱۸ Fully Associative (FA)

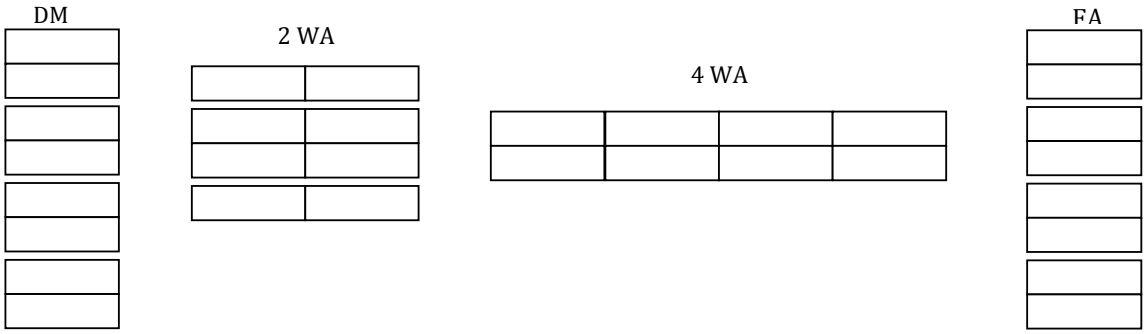
صندلی‌های زوج با آن شماره می‌نشانیم. اگر هر دو صندلی آن زوج، پر بودند، یکی از دانشجویان در آن زوج را بیرون می‌اندازیم. این همان روش 2WSA است.

- هر دانشجوی جدیدی که وارد شد، روی هر صندلی که خالی بود بنشیند و اگر صندلی خالی موجود نیست یکی از افراد قبلی را بیرون می‌اندازیم. این همان مکانیزم FA است.

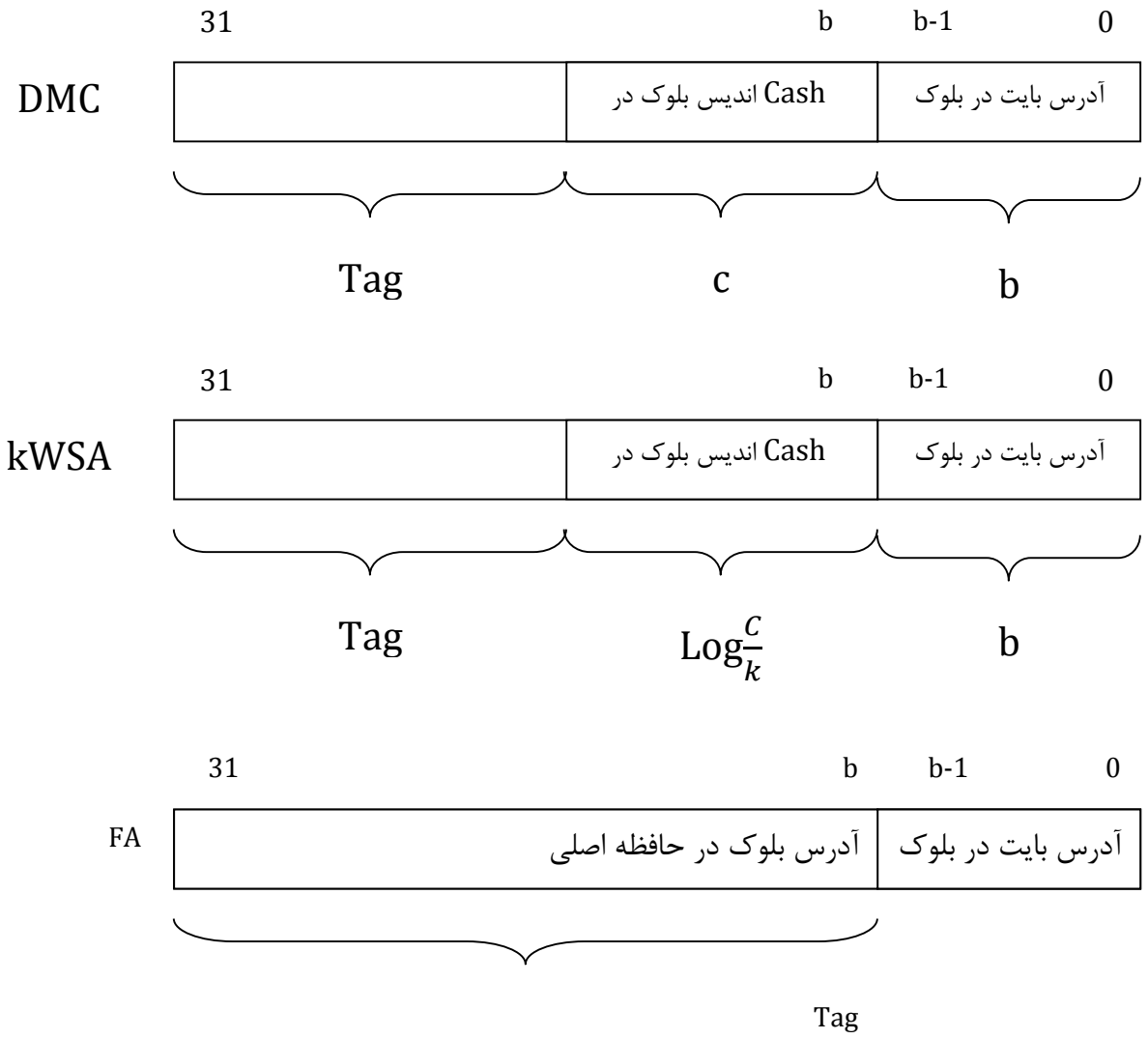
طبیعی است اگر فرایند بیرون انداختن دانشجو در کلاس به صورت هوشمندانه‌ای باشد، راه‌حل سوم Hit Ratio بالاتری دارد. البته در راه‌حل سوم دسترسی به دانشجو زمان بیشتری می‌گیرد. در واقعیت داریم:



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:

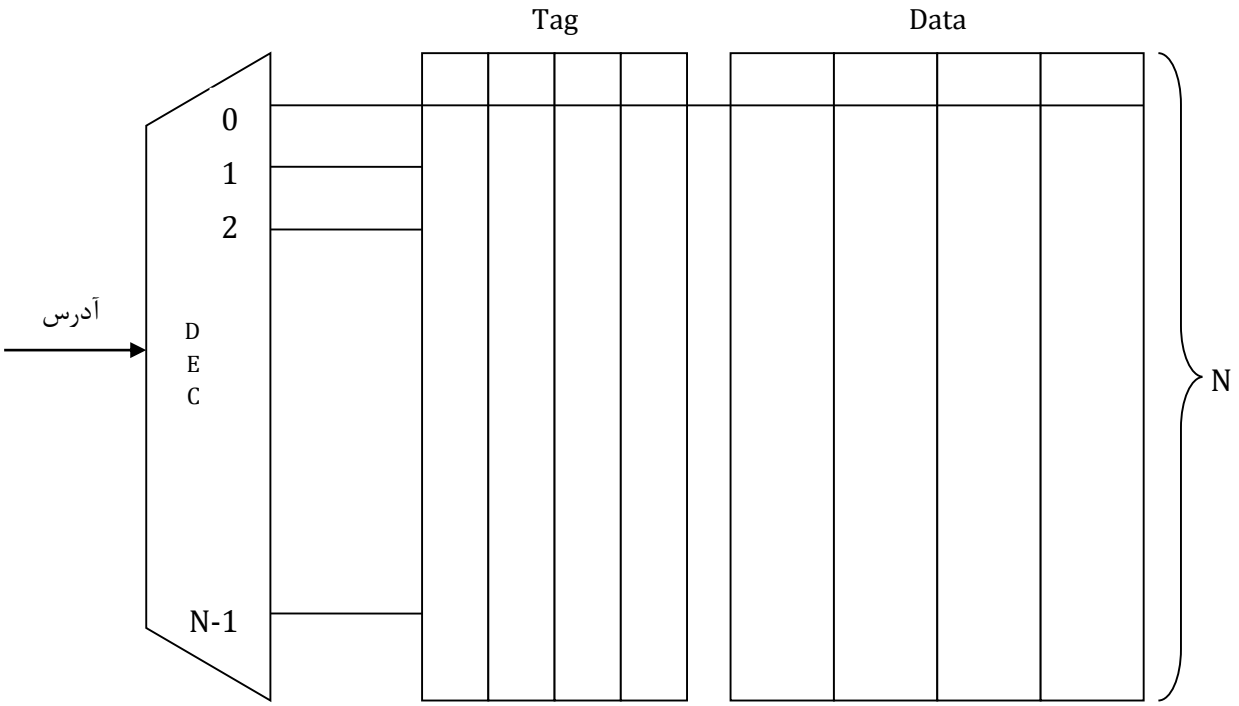


پ) طراحی مداری حافظه‌های نهان:

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 شده سایز برچسب، توان مصرفی و مساحت این نوع حافظه‌ی نهان به شدت زیاد بوده و مقرون به صرفه نیست.

سیاست جایگزینی

مسلماً بهترین سیاست‌ها، سیاست‌هایی‌اند که المان‌هایی را اضافه و حذف کند که در آینده به نفعمان باشد. هدفمان در این بخش این است تا بررسی کنیم چه سیاست‌هایی برای جایگزینی داده‌ها در حافظه‌ی نهان مناسب‌ترند. برای مثال در یک 4WSA، اگر قرار باشد عنصری جدید وارد حافظه‌ی نهان کنیم ولی مجموعه‌ای که می‌خواهیم داده را به آن اضافه کنیم پر باشد، کدامیک از چهار عنصر مجموعه را بیرون بیندازیم بهتر است؟

روشن است که بحث سیاست 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 بیشتر شود این مکانیزم پیچیده‌تر و در عین حال مهم‌تر خواهد شد. انواع روش‌هایی که برای جایگزینی می‌توان تصور کرد عبارتند از:

- Random: عنصری را به تصادف از مجموعه بیرون بیندازیم.
- FIFO[۱۹]: هر عنصری که دیرتر وارد شده را بیرون بیندازیم.
- LIFO[۲۰]: هر عنصری که زودتر وارد شده را بیرون بیندازیم.
- LRU[۲۱]: عنصری که اخیراً کمتر استفاده شده بیرون برود.
- MRU[۲۲]: عنصری که اخیراً بیشتر استفاده شده بیرون برود.
- LFU[۲۳]: عنصری که تا به حال کمتر استفاده شده بیرون برود.
- MFU[۲۴]: عنصری که تا به حال بیشتر استفاده شده بیرون برود.

در روش LFU نیاز به شمارنده 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 پیاده‌سازی 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.

---

۱۹ First In First Out
۲۰ Last In First Out
۲۱ Least Recently Used
۲۲ Most Recently Used
۲۳ Least Frequently Used
۲۴ Most Frequently Used

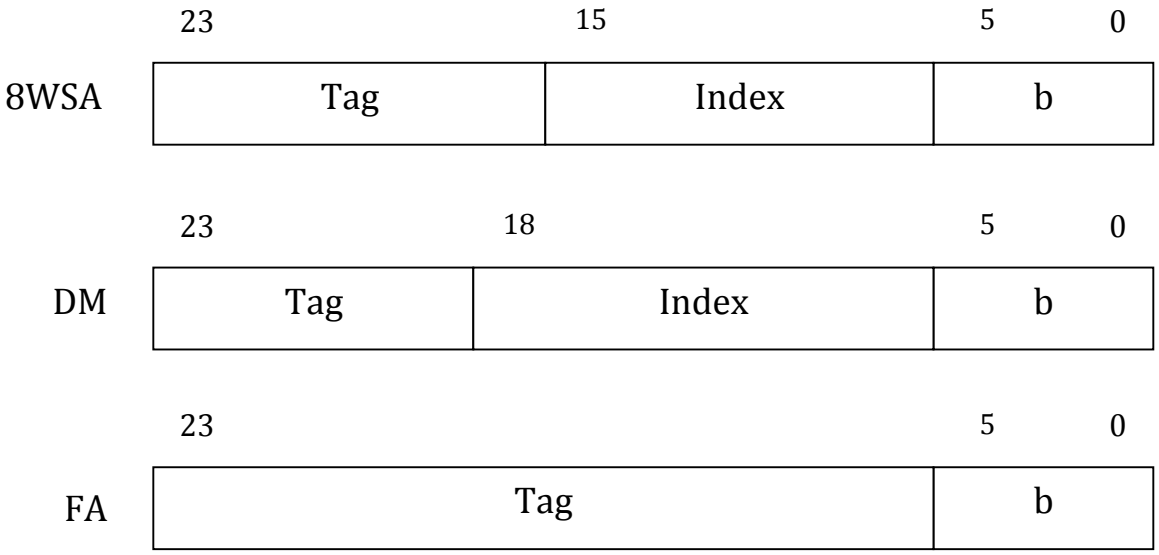
پس از LRU، Random کارایی نسبتاً خوبی دارد و در حالتی که نمی‌خواهیم سخت افزار اضافه‌تر برای Rank بگیریم مناسب است.

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

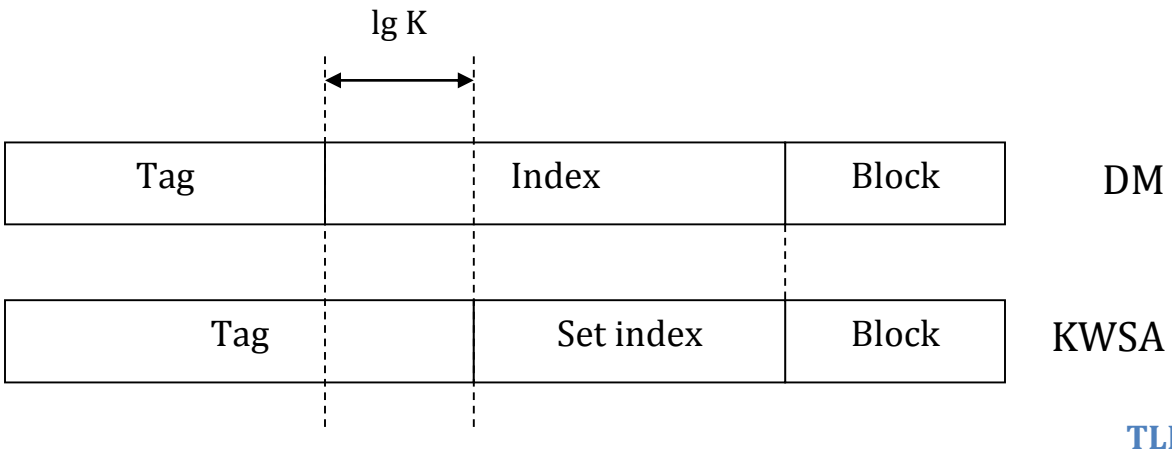
مثال: چنانچه حافظه‌ی نهانی به اندازه‌ی 512KB و اندازه‌ی بلوک 64B داشته باشیم و اندازه‌ی حافظه‌ی اصلی 16MB باشد، مطلوب است اندازه و قالب آدرس برای هر یک از حالات زیر:

8WSA
DM
FA

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



در حالت کلی:



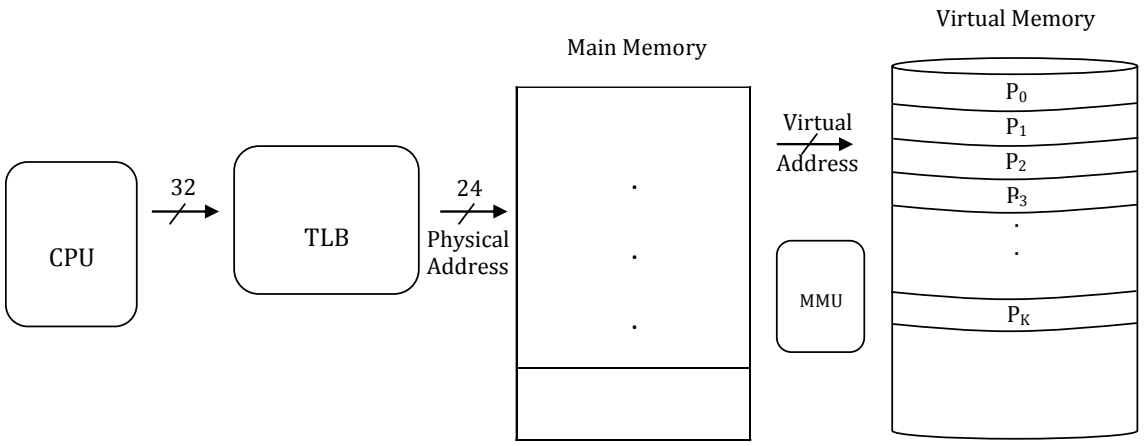
TLB

وقتی CPU ۳۲ بیت آدرس میدهد و حافظه ۱۶MB و باس آدرس ۲۴ بیتی است استفاده از ۲۴ بیت کم ارزش کارایی بالایی به ما نمیدهد. به همین دلیل هارد دیسک مطرح و از آنجایی که هارد دیسک خیلی کند و لخت است از آن به صورت یک Virtual Memory در کنار Main Memory استفاده شد. این امر به طور کلی دو مزیت دارد:

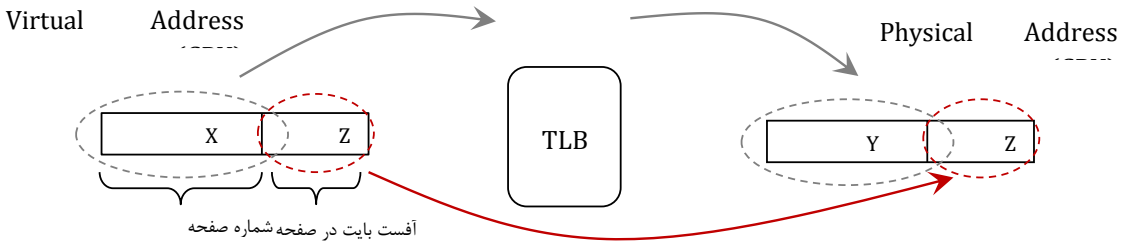
- استفاده از کل ۳۲ بیت
- سرعت نسبتاً بالا

به این منظور هارد دیسک و Main Memory به واحد هایی به نام page تقسیم شدند که از آن برای آدرس دهی استفاده شد. به علاوه اندازه هر page وابسته به عوامل مختلفی از جمله سرعت دیسک ،. .. میباشد.

P'_0
P'_1
P'_2



از آنجا که آدرسی که از طرف CPU فرستاده میشود ۳۲ بیتی و Main Memory ۲۴ بیتی است از TLB به عنوان مترجم استفاده میشود. به این صورت که آدرسVirtual از سمت CPU را به آدرس فیزیکی در Main Memory ترجمه میکند (شماره page اصلی را گرفته و شماره page در Main Memory را میدهد).



یک TLB(Translation Lookaside Buffer) در واقع یک حافظه میانجی در CPU است که قسمتهایی از جدول نگاشت pageها را برای ترجمه آدرس مجازی به فیزیکی در خود نگاه میدارد.

این جدول میتواند به عنوان مثال به شکل زیر باشد:

P	P’
0	2
1	0
2	X
3	X
.	.
.	.
.	.
232-1	1

اما در این حالت بسیاری از خانه‌ها معتبر نیستند. برای حل این مشکل و بهینه سازی، جدول را برعکس کرده و به صورت زیر تغییر دادند تا فضای کمتر و بهینه تری را اشغال کند.

P	P’
1	0
99	1
0	2
2	3
.	.
.	.
.	.
6	K

یک TLB نوعا یک حافظه CAMاست و بر اساس محتوا جستجو کرده و این گونه عمل میکند که آدرس واقعی دریافت شده را با تک تک pageها مقایسه میکند و اگر برابر بودند یک hit رخ میدهد و آدرس فیزیکی را بر میگرداند و در غیر این صورت (داده در Main Memory نباشد) miss رخ داده است.
TLB در مواردی نیز به دلیل توان مصرفی بالای حافظه‌های CAMبا مکانیزم k-way set associative مورد استفاده قرار میگیرد.

در حالتی که miss رخ میدهد MMU که معمولاً در خود TLB جا دارد، داده مورد نظر را با استفاده از الگوریتم LRU(Least Recently Used) در Main Memory جایگزین میکند.

به عبارتی میتوان گفت که TLB تقریباً شبیه یک Fully Associative Cache عمل میکند که خانه‌های آن به tagهای موجود در cache شباهت دارند.

و اما مسئله ای دیگر محل cache در کنار TLB است. اگر cache قبل از TLB قرار بگیرد به این معنی خواهد بود که داده‌های هارد دیسک را در بر دارد که در این حالت کارایی بسیار بالا میرود ولی بزرگتر شدن آدرس‌ها باعث ایجاد هزینه بیشتر نیز میشود.

و اگر بعد از TLB باشد به این معنی خواهد بود که داده‌های مجود در Main Memory را در خود نگاه میدارد که این بار کارایی و هزینه حالت قبل را نخواهد داشت.

به علاوه با توجه به اینکه ترکیب متوالی cache و TLB وقت گیر است این دو را به صورت موازی در کنار هم به کار میگیرند. با این حال در مواردی نیز cache قبل و اکثراً بعد از TLB قرار میگیرد.

برآورد کارایی[۲۵]

علم حیل، همان علمی است که راه های شناخت تدابیر و شیوه های عملی کردن مفاهیم ریاضی در صنعت را مشخص می سازد و نشان می دهد که چگونه می توان مفاهیم عقلی ریاضی را در اجسام طبیعی آشکار نمود. .. یکی از علوم حیل(=مهندسی)، علمی است که پیرامون ساختن ابزار و وسایل برای صنایع عملی مورد استفاده قرار می گیرد.

تفاوت کارایی و بازدهی عملیاتی

ما در این مجال، تفاوتی بین کارایی(Performance) و بازدهی عملیاتی(Throughput) قایل نیستیم. زیرا در صفحات آینده خود را به ریزپردازنده و کارایی تعریف شده برای آن محدود می کنیم. لیکن می توان گفت بازدهی عملیاتی(Throughput) زیر شاخه ای از کارایی(Performance) است.

- کارایی(Performance): اصطلاحی کلی است. می توان آن را ایده آل کردن فرآیند در سیستم‌ها دانست. موضوع این ایده آل سازی می تواند اجزا و مشخصات گوناگون سیستم باشد: گاهی زمان پاسخ سیستم به یک درخواست(Execution Time)، گاهی Clock، گاهی سرعت، گاهی توان مصرفی (Power) و. .. .

تعریف غیر دقیق بازدهی عملیاتی طبق کتاب پترسون به شرح زیر است:

- بازدهی عملیاتی (Throughput): مجموع کاری که یک سیستم در یک زمان مشخص می تواند انجام دهد.

به عنوان مثال مدیر یک شبکه علاقه مند است تا بازدهی عملیاتی سرور بالا باشد تا درخواست های بیشتری را در طول روز به سرانجام برساند.

---

۲۵ Performance Evaluation

کنفرانس های جهانی

کنفرانس هایی هم در دنیا داریم که تنها پیرامون کارایی سیستم های کامپیوتری و دست یابی به کارایی بالا (High Performance) است. از آن جمله است :

1. HPCA (High Performance Computer Architecture)
2. ISCA
3. MICRO

هر وقت به دنبال اطلاعات به روز هستید، به کتابچه های این‌ها مراجعه کنید.

تبیین اهمیت موضوع

همان طور که در سخن فارابی اشاره شده بود، هنر مهندسی پیاده سازی مدل های ذهنی در جهان بیرونی است. مهندسی کامپیوتر هم این گونه است. حال فرض کنید برای مدلی که از یک کامپیوتر در ذهنمان داریم، ۲ نمونه ساخته شده است. سوال اساسی این جاست "کدام یک از این ۲ رایانه، مطلوب تر است؟". مطلوب بودن یک کامپیوتر کاملا بستگی به کاربر و کاربری مورد انتظار او از کامپیوتر دارد. لذا نکته مهم این است که کدام یک برای "کارِ" کاربر، مناسب تر است. ما از شاخص کارایی(Performance) برای برآورد این "مناسب بودن" و "مقایسه" دو دستگاه در این زمینه، استفاده می کنیم.

کاربر	کار مورد انتظار از رایانه	مفهوم کارا بودن رایانه برای کاربر	نگرانی کاربر پیرامون قیمت رایانه
سازمان هواشناسی	پردازش حجم زیادی از داده‌ها	قدرت پردازشی رایانه	نگران نیست
سازمان هوافضا	کنترل باله های موشک	سریع، دقیق و بی درنگ عمل کردن	نگران نیست
گرافیست	کارهای گرافیکی سنگین	از پس تولید جلوه های بصری و تصاویر، در زمان کوتاهی برآید.	کمی نگران
کاربر خانگی	بازی، مرور اینترنت و برخی نیازهای روزمره	از هر نظر نسبتا قابل قبول عمل کردن(سرعت، قدرت محاسباتی و. ..)	کاملا نگران

جدول ۲ – مثالی از تفاوت مفهوم کارایی در زمینه های مختلف

اما نکته ای که در نظر تمامی کاربران اهمیت دارد، این است که رایانه نسبتاً "سریع" عمل کند. لذاست که می توان گفت که کارایی با هر تعریفی که بیان شود در رابطه زیر صدق می کند:

$$\text{Performance} \sim \frac{1}{\text{Execution time}}$$

که تعریف زمان اجرا(Execution time) که گاهی از آن با Response Time یاد می شود، طبق تعریف کتاب پترسون به قرار زیر است:

- زمان اجرا (Execution Time): مجموع زمانی که یک رایانه نیاز دارد تا یک وظیفه کاری(Task) را به سرانجام برساند. شامل: زمان اجراشدن روی CPU، زمان دسترسی به حافظه، زمان صرف شده توسط سیستم عامل برای پاسخ گویی به درخواست های برنامه و. .. .

ما فعلا از رابطه بالا برای بررسی کارایی استفاده می کنیم. در انتهای این بخش، عامل هزینه(cost) را هم برای تحلیلی واقع گرایانه تر در نظر خواهیم گرفت.

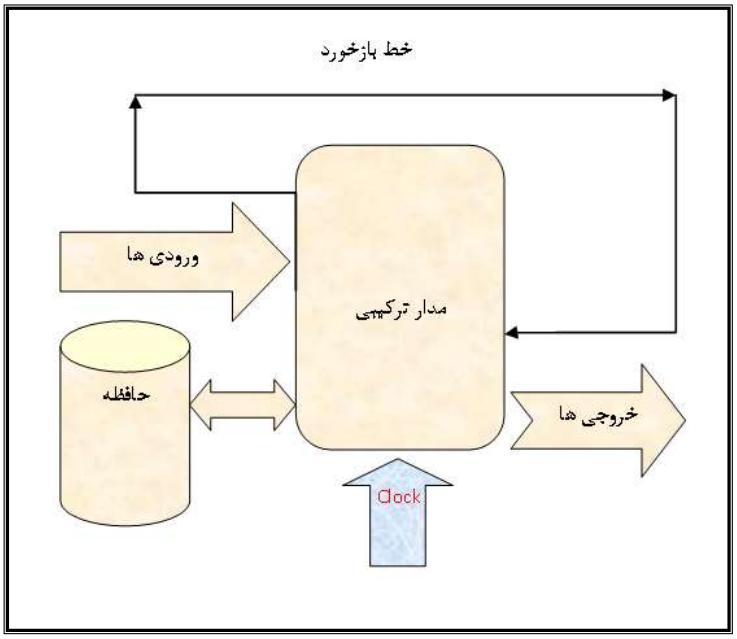
مثال) برنامه X روی کامپیوتر A و B به ترتیب ۱۰ و ۱۵ ثانیه طول می کشد. مشخصات دیگر رایانه‌ها را یکسان در نظر بگیرید. مشخص کنید که کامپیوتر A چند برابر B کاراتر است؟

پاسخ:

$$\frac{\text{Performance A}}{\text{Performance B}} = \frac{\text{Execution Time B}}{\text{Execution Time A}} = \frac{15}{10} = 1.5$$

وابستگی زمان اجرا به عوامل

می دانیم تقریبا تمامی کامپیوتر های متداول امروزی، بر اساس مدل زیر عمل می کنند:



لذا می توان گفت که هر عملیات طی چندین کلاک انجام می شود. اگر فرکانس کاری ریزپردازنده برابر f باشد آن گاه هر کلاک، زمانی برابر $\frac{1}{f}$ نیاز دارد. لذا داریم :

$$\text{Execution Time (زمان اجرای برنامه)} = \frac{\text{تعدادکلاک های مصرفی برنامه}}{f}$$

می دانیم که هر برنامه از چندین دستور تشکیل شده است. اگر تعداد دستورات برنامه n باشد داریم:

$$\text{Execution Time (زمان اجرای برنامه)} = \sum_{i=1}^{n} \frac{\text{تعدادکلاک های مصرفی دستورiام}}{f}$$

CPI

CPI (Clock per Instruction) یک پارامتر پرکاربرد در بررسی کارایی (Performance) یک رایانه است. بنا به تعریف برای مجموعه دستور العمل X با k دستور داریم:

- تعداد کلاک به ازای دستور در مجموعه دستورالعمل های X :

$$CPI_X = \frac{\text{تعداد کلاک های مصرفی برای همه } K \text{ دستور مجموعه } X}{K}$$

این پارامتر را برای ۴ مجموعه از دستور العمل‌ها محاسبه می کنند:

۱. یک تک دستور از مجموعه دستورات ریزپردازنده(Instruction set):

$$CPI_i = \text{تعداد کلاک لازم برای اجرای دستور i ام}$$

در این حالت، به مفهوم اندیس CPI دقت شود.

۲. مجموعه تمامی دستورات ریزپردازنده (Instruction Set). در این صورت :

$$CPI = \frac{\sum_{i=1}^{\text{تعداد دستورات } instruction\ set} \text{تعداد کلاک مصرفی برای دستور } i \text{ ام}}{\text{تعداد دستورات } instruction\ set} = \frac{\sum_{i=1}^{\text{تعداد دستورات } instruction\ set} CPI_i}{\text{تعداد دستورات } instruction\ set}$$

۳. مجموعه دستورات یک برنامه خاص :

$$CPI = \frac{\sum_{i=1}^{\text{تعداد دستورات برنامه}} \text{تعداد کلاک مصرفی برای دستور } i \text{ ام}}{\text{تعداد دستورات برنامه}}$$

۴. کلاس j ام از مجموعه دستورات ریزپردازنده(Instruction set) :

$$CPI_j = \frac{\sum_{i=1}^{\text{تعداد دستورات کلاس } j \text{ ام } instruction\ set} \text{تعداد کلاک مصرفی برای دستور } i \text{ ام این کلاس}}{\text{تعداد دستورات کلاس } j \text{ ام } instruction\ set}$$

در این حالت، به مفهوم اندیس CPI دقت شود.

اکنون که با CPI آشنا شدیم می توانیم با رابطه زیر نیز آشنا شویم:

$$\text{زمان اجرای برنامه(Execution Time)} = \sum_{k=1}^{t} \frac{C_k . CPI_k}{f}$$

t تعداد کلاس های دستورات ریزپردازنده است.

C_k تعداد دستورات برنامه که متعلق به کلاس k ام دستورات ریزپردازنده هستند، است.

CPI_k نرخ کلاک به ازای دستور برای مجموعه دستورات کلاس k ام دستورات ریزپردازنده است.

نکته: در این مبحث، تمام دستوراتی از مجموعه دستورالعمل‌ها که تعداد کلاک یکسانی مصرف می کنند در یک کلاس از Instruction Set قرار می گیرند.

نکته: همان طور که گفتیم، زمان اجرا را فقط ریزپردازنده تعیین نمی کند. عواملی مانند حافظه سیستم، سیستم عامل و. .. نیز در زمان اجرا موثر اند. مثلا دستورات باید از حافظه واکشی (Fetch) شوند. لذاست که تعداد کلاک مصرفی برنامه به آن عوامل نیز مربوط می شود. برای همین می توان گفت که فرمول های بالا بـا کمـی تسـامح بیان شده اند.

IPC

IPC (Instruction per Clock) پارامتر مشابهی است گه گاه به گاه مورد استفاده قرار می گیرد. تعریف آن بـه قرار زیر است:

- نرخ تعداد دستور انجام شده به ازای یک کلاک در مجموعه دستورالعمل های X :

$$IPC_X = \frac{K}{\text{تعداد کلاک های مصرفی برای همه } K \text{ دستور مجموعه } X}$$

مشابه آن چه در مورد CPI گفتیم، مجموعه X می تواند ۴ حالت گوناگون داشته باشد که در بخش قبـل بحـث شد. رابطه زیر ۲ مفهوم بیان شده اخیر را به هم پیوند می زند:

$$CPI \times PCI = 1$$

نکته: CPI و IPC بدون واحد هستند.

سه پارامتر اساسی

برای تعیین کارایی یک سیستم به زمان اجرای برنامه روی آن روی آوردیم. سوالی که مطرح است این است کـه چه پارامترهایی در تعیین این زمان موثر است. مطابق روابط صفحات قبل داریم :

$$\text{زمان اجرای برنامه (Execution Time)} = \frac{\text{تعداد دستورات} \times CPI}{f}$$

لذاست که می توان گفت کارایی یک سیستم کامپیوتری به ۳ عامل زیر مرتبط است:

۱. CPI دستورات برنامه که ساختمان و نحوه طراحی ریزپردازنده تعیین کننده آن است.

۲. تعداد دستورات برنامه که Instruction Set و الگوریتم کامپایل کردن برنامه تعیین کننده آن است.

۳. فرکانس کاری ریزپردازنده.

اگر در تحلیل خود از کارایی سیستم‌ها، هر کدام از این ۳ عامل را در نظر نگیریم، ره به جایی نخواهیم برد. لذا نباید فریب بهینه بودن تنها یکی از موارد بالا را خورد. ممکن است از خود بپرسید که "پس چرا در بازار تنها روی فرکانس کاری ریزپردازنده تبلیغ می شود؟" پاسخ این جاست که شرکت های سازنده معمولا سازگاری (compatibility) یک نسل از ریزپردازنده‌ها را رعایت می کنند و از سویی برنامه های کاربردی و سیستم های عامل مطابق با اشتراکات موجود بین Instruction set ریزپردازنده های یک نسل نوشته می شوند، لذا عملا پارامترهای "CPI" و "تعداد دستورات برنامه" برای حجم زیادی از برنامه هایی که کاربر خانگی یا اداری اجرا می کند، به ازای ریزپردازنده های مختلف یکسان است. لیکن یک تحلیل جامع باید این ۲ پارامتر را نیز در نظر بگیرد.

MIPS

MIPS (Million Instruction per Second) مقیاسی دیگر برای مقایسه کارایی ۲ سیستم است.

$$\text{MIPS} = \frac{1}{10^6} \times \frac{\text{تعداد دستورهای اجرا شده}}{\text{زمان طی شده به ثانیه}}$$

واحد این پارامتر را هم معمولا MIPS می نامند. اگر دستورات اجرا شده روی ۲ سیستم دقیقاً یکی باشد، آن سیستمی که MIPS بیشتری دارد، کاراتر است. لیکن اگر دستورات متفاوت باشد هیچ قضاوتی نمی توان کرد.

رابطه زیر را برای MIPS داریم:

$$\text{MIPS} = \frac{\text{تعداد دستورات اجرا شده}}{Execution\ Time \times 10^6} = \frac{f}{\text{CPI} \times 10^6}$$

تمرین ۵: MIPS پردازنده خود را حساب کنید.

نکته: همان گونه که گفتیم اگر دستورالعمل های اجرا شده یکسان نباشند می توان مثالی زد که ریزپردازنده ای که دارای MIPS بالاتری باشد، دارای زمان اجرای بدتری باشد. لذا در حالت کلی رابطـه MIPS و کـارایی لزومـاً مستقیم نیست.

مثال اول کارایی :

برنامه ای در کامپیوتر A در ۱۰ ثانیه اجرا می شود. نرخ کلاک در A برابـر ۴۰۰ مگـاهرتز است. در کامپیوتر B در ۶ ثانیه اجرا می شود. نرخ کلاک B را بدست آورید. درضمن مـی دانیم که برای هر دستور، کامپیوتر B به ۱.۲ برابر تعداد کلاک لازم روی A به کلاک نیاز دارد.

$$1.2 \times CPI_A = CPI_B$$

تعداد کلاک لازم $= 1.2 \times Instruction\ Count \times CPI_A = B$

$$6 = \frac{(InstructionCount.CPI_B)}{f_B} = \frac{(InstructionCount.CPI_A) \times 1.2}{f_B} = \frac{f_A \times 10 \times 1.2}{f_B}$$

$$f_B = 800 \times 10^6\ Htz$$

مثال دوم کارایی :

یک برنامه بر روی ۲ رایانه با اجرا شده است. از آن جایی که کامپایلرها متفاوت بوده است، ۲ کد مختلف به دست آمده است. اگر کلاس های دستورات A , B,C دارای CPI زیر باشند و تعداد دستورات از هر کلاس در ۲ کد مختلف به قرار زیر باشد مطلوب است رایانه کاراتر را بیابید. هم چنین رایانه ای که دارای MIPS بالاتری است را بیابیـد. آیـا ایـن ۲ یکسـان هستند؟ فرکانس کاری هر دو رایانه را ۱ مگاهرتز فرض کنید.

	تعداد دستور کلاس A	تعداد دستور کلاس B	تعداد دستور کلاس C
کد برنامه روی ماشین ۱	۵	۱	۱
کد برنامه روی ماشین ۲	۱۰	۱	۱

کلاس دستور	CPI
A	1
B	2
C	3

$$\text{MIPS1}=\frac{\text{f}}{CPI\ OF\ PROGRAM \times 10^6}=\frac{1}{\frac{5\times1+1\times2+1\times3}{1+1+5}}=0.7$$

$$\text{MIPS2}=\frac{\text{f}}{CPI\ OF\ PROGRAM \times 10^6}=\frac{1}{\frac{10\times1+1\times2+1\times3}{10+1+1}}=0.8$$

$$\frac{1}{\text{Execution time}}\sim \text{Performance}$$

$$\text{Performance 1}\sim\frac{\text{f}}{\text{Instruction Count}\times\text{CPI}}=\frac{\text{f}}{7\times\frac{5\times1+1\times2+1\times3}{1+1+5}}=10^5$$

$$\text{Performance 2}\sim\frac{\text{f}}{\text{Instruction Count}\times\text{CPI}}=\frac{\text{f}}{12\times\frac{10\times1+1\times2+1\times3}{10+1+1}}=0.07\times10^5$$

لذا کامپیوتر ۱ کاراتر است و کامپیوتر ۲ دارای MIPS بیشتری است. لذا در این مثال دیدیم که MIPS با کارایی لزوماً نسبت مستقیم ندارد.

دو سبک طراحی

در ابتدای اختراع رایانه، رایانه‌ها دارای مجموعه دستورالعمل های اندک و ساده ای بودند. با رشد فناوری رایانه و همه گیر شدن آن، طراحان به سمت رایانه هایی با مجموعه دستورالعمل های پیچیده رفتند. لیکن پس از چندین سال متوجه شدند که این پیچیدگی تبعاتی چند به همراه دارد:

- پروسه طراحی را پیچیده و هزینه بر می کند.
- زمان ارایه محصول به بازار (Time To Market) را افزایش می دهد. امری که سبب عقب افتادن شرکت موتورولا از شرکت اینتل در بازار ریزپردازنده های رایانه های شخصی در قرن گذشته میلادی شد.
- عیب یابی مدارات و تضمین صحت عملکرد آن را برای کاربرد های حیاتی(Critical) مشکل می سازد.

لذا طراحان دوباره به سمت مجموعه دستورالعمل های اندک و ساده بازگشتند. این بازگشت ابتدا توسط محافل دانشگاهی در دهه اواخر دهه ۸۰ و ابتدای دهه ۹۰ میلادی رخ داد.

RISC & CISC

CISC (Complex Instruction Set Computer) و RISC (Reduced Instruction Set Computer)

نام ۲ دسته از رایانه هاست که دارای ایده های معماری متفاوتی هستند. اگرچه هردو دسته از الگوریتم فون نیومن بهره می برند لیکن در مشخصات زیر با هم متفاوت اند :

نوع مشخصه	مشخصه RISC	مشخصه CISC	ملاحظه
تعداد و تنوع دستورات در مجموعه دستور العمل	کم	زیاد	
تعداد کلاک مصرفی برای دستورات مختلف	ثابت و کم	متغیر و زیاد	برای همین در RISCها می توان پهنای کلاک را کاهش داد.
شیوه های آدرس دهی	محدود و کم	زیاد و متنوع	معمولا انواع آدرس دهی های غیر مستقیم را در RISCها نداریم.
تعداد بایت مصرفی هر دستور	ثابت و کم	متغیر و گاهی زیاد	
تعداد ثبّات های عمومی	کم	زیاد	مثلا برای دستور جذر در یک رایانه CISC به چندین و چند ثبّات میانی نیاز است.
تعداد عملوند های دستورالعمل	ثابت و کم	زیاد و متغیر	
تعداد دستورات لازم برای یک برنامه مشخص	بیشتر از CISC	کمتر از RISC	

البته گاهی هر دوی این دیدگاه‌ها را می توان در یک ریزپردازنده جست و جو کرد. مثلا هسته پنتیوم توسط دیدگاه RISC ساخته شده است لیکن سعی شده است که کل ریزپردازنده از بیرون همانند CISC به نظر آید.

نکته: در عبارات مربوط به محاسبه CPI، در رایانه های RISC کران عبارت Σ افزون تر است ولی پهنای کلاک کمتری دارد.

نکته: در عبارات مربوط به محاسبه CPI، در رایانه های CISC کران عبارت Σ کوچک تر است ولی پهنای کلاک بیشتری دارد.

قانون ۸۰-۲۰

این قانون شهودی و حدودی روی بسیاری از پدیده‌ها حکم فرماست. مثلا شما با ۲۰ درصد از تلاش آرمانی ممکن است تا ۸۰ درصد نتیجه نهایی را بگیرید. یا ۲۰ درصد مجموعه دستورالعمل های یک کامپیوتر، نیاز ۸۰ درصد کاربران را برآورده می کند.

قانون آمدال

آقای آمدال _که سرپرستی پروژه IBM mainframes را چندی به عهده داشت اولین بار این قانون را ارایه کرد. این قانون به ما کمک می کند تا دریابیم که روی کدام بخش از یک سیستم باید سرمایه گذاری بیشتری بکنیم تا بازده بیشتری بدست آوریم.

- قانون آمدال: اگر نسبت دستورات ترتیبی یک برنامه به کل دستورات آن f باشد(یعنی نتوان در اجرای آن بخش تسریع کرد) و الباقی برنامه را بتوان_به هر طریقی_ p برابر سریع تر اجرا کرد، میزان افزایش سرعت برنامه(Speed-Up)، یعنی نسبت زمان اجرا در حالت دوم به حالت اول، از رابطه زیر به دست می آید:

$$\text{Speed-Up} = \frac{1}{f + \frac{1-f}{p}} = \frac{p}{(p-1).f + 1}$$

نکته: حالت تعمیم یافته قانون آمدال؛ اگر a_i نسبت از برنامه را بتوان به میزان p_i برابر سریع تر اجرا کرد :

$$\text{Speed-Up} = \frac{1}{(1-\sum_{i=1}^{n} a_i) + \sum_{i=1}^{n} (\frac{a_i}{p_i})}$$

تمرین ۵: قانون آمدال را اثبات کنید.

مدت زمان اجرای برنامه قبل از افزایش سرعت را x می نامم. مدت زمان اجـرای برنامـه بعـد از افزایش سرعت را y می نامم.

این گونه نسبت افزایش سرعت برابر مقدار زیر است:

$$speedUp = \frac{x}{y}$$

ازسویی دیگر طبق فرض مساله داریم:

$$x = x.f + x.(1-f)$$

که جمله اول عبارت بالا بخشی است که سرعت آن تغییر نخواهد کرد و قسمت دوم عبارت بالا بخشی است که تغییر خواهد کرد.

پس از تغییر سرعت، بخش دوم در $\frac{1}{p}$ زمان قبل اجرا می شود لذا برای y داریم:

$$Y = x.f + (\frac{x.(1-f)}{p})$$

لذا برای نسبت $\frac{x}{y}$ داریم :

$$speedUp = \frac{x}{y} = \frac{x.f + x.(1-f)}{x.f + (\frac{x.(1-f)}{p})} = \frac{f + 1 - f}{f + \frac{1-f}{p}} = \frac{1}{f + \frac{1-f}{p}}$$

مثال اول آمدال:

یک برنامه روی یک کامپیوتر در ۱۰۰ ثانیه اجرا می شود که ۶۰ ثانیه آن مربوط به دستورالعمل های ضرب برنامه است. دستورالعمل های ضرب چقدر سریع تر شوند تا اجرای برنامه ۲,۵ برابر سریعتر گردد؟

$$Speed\text{-}Up = 2.5 = \frac{1}{f + \frac{1-f}{p}} = \frac{1}{.4 + \frac{.6}{p}} \rightarrow 0 = 1 \text{ (تناقض)}$$

پس چنین کاری امکان ندارد. راه حل دیگر این بود که اگر برنامه بخواهد 2.5 برابر سریعتر شود یعنی باید در $\frac{1}{2.5} \times 100 = 40$ ثانیه اجرا شود. لیکن می دانیم که دستورات غیر از ضرب ۴۰ ثانیه زمان می خواهند. این یعنی باید سرعت اجرای بخش ضرب بی نهایت باشد تا هیچ زمانی نبرد. اما این موضوع امکان ندارد.

مثال دوم آمدال :

یک برنامه در زمان ۸۰ ثانیه بر روی یک رایانه اجرا شده است. ۲۰ درصد زمان برای دستورات ممیز شناور و ۳۰ درصد زمان برای دستورات ضرب اعداد صحیح مصرف شده است. اگر اجرای دستورات ممیز شناور را ۸ برابر و اجرای دستورات ضرب اعداد صحیح را ۶ برابر تسریع کنیم، میزان تسریع برنامه چقدر است؟

مطابق تعمیم قانون آمدال داریم:

$$Speed\text{-}Up = \frac{1}{(1-.3-.2) + \frac{.3}{6} + \frac{.2}{8}} = 1.74$$

مثال سوم آمدال :

تابع ریشه دوم اعشاری در یک برنامه گرافیکی به طور معمول به کار می رود. فرض کنید زمان اجرای این تابع، ۲۰ درصد زمان از زمان اجرای برنامه گرافیکی را مصرف کند. ۲ راه کار برای بهبود برنامه موجود است. پیشنهاد اول: تابع ریشه دوم را ۱۰ برابر سریع تر کنیم.

پیشنهاد دوم: همه دستورات ممیز شناور را ۲ برابر تسریع کنیم.

این عملیات اعشاری ۵۰ درصد زمان کار گرافیکی را به خود مشغول می کند. کدام راه کار بهتر است؟

پیشنهاد اول: $Speed\text{-}Up = \frac{1}{(1-.2)+\frac{.2}{10}} = 1.22$

پیشنهاد دوم: $Speed\text{-}Up = \frac{1}{(1-.5)+\frac{.5}{2}} = 1.33$

لذا راه کار دوم بهتر است.

بستر آزمایشBenchmark

گفتیم که کارایی یک سیستم را معمولا با اجرای یک برنامه روی آن اندازه می گیریم. ولی نگفتیم چه برنامه ای. مسلماً برای عادلانه بودن قضاوت ما باید برنامه های آزمایشگر برای همه سیستم های هم ردیف، یک سان باشد. برنامه های بستر آزمایش این وظیفه را به عهده دارند. برای سرورها برنامه های بستر آزمایش ویژه ای داریم. برای کامپیوتر های نهفته(مانند موبایل ها) برنامه های آزمایش ویژه خودشان راداریم_مانند MIBENCH_ و بالاخره برای پردازنده های رایانه های شخصی PC نیز برنامه های بستر آزمایش ویژه ای وجود دارد_مانند SPEC CPU 2000.

البته این برنامه‌ها به ۲ دسته زیر تقسیم می شوند:

۱. CINT که عملیات اعداد صحیح را انجام می دهد.

۲. CFP که عملیات اعداد اعشاری انجام می دهد.

فصل دوم

ALU

همانطور که می‌دانید طراحی یک CPU، شامل بخش‌های متفاوتی است. یکی از این بخش‌ها واحد منطق و محاسبات است. در این بخش قصد داریم که این واحد را مورد بررسی قرار دهیم و در نهایت آن را طراحی کنیم.

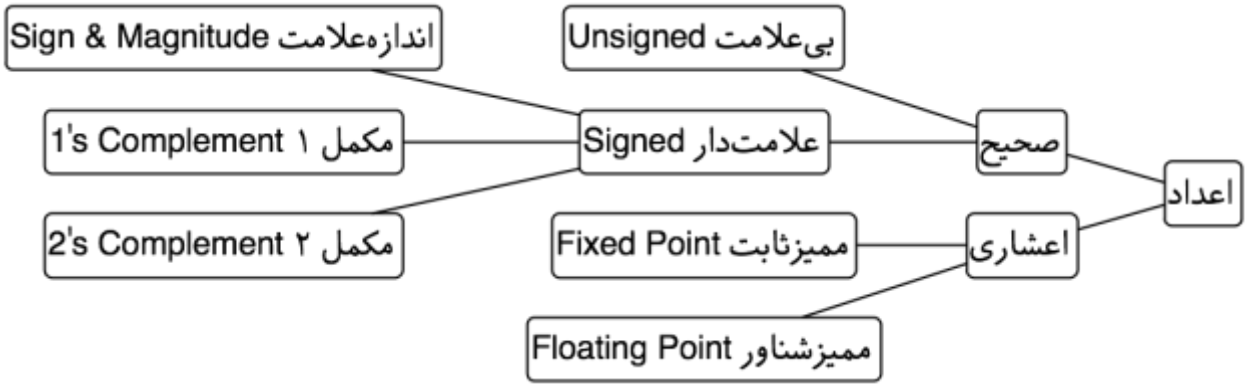
در ابتدا با جمع کننده‌ها آشنا می‌شویم، سپس می‌خواهیم عملیات ضرب را در پردازشگر طراحی کنیم. در این مرحله ابتدا با ضرب کننده ترتیبی آشنا می‌شویم، که برای سرعت بخشیدن به آن از الگوریتم بوث استفاده می‌شود. سپس با نحوه‌ی اجرای عملیات تقسیم آشنا خواهیم شد.

در پیاده سازی اعداد اعشاری ابتدا از شیوه ممیز ثابت استفاده می‌کنیم، اما می‌بینیم استفاده‌ی ما بسیار محدود می‌شود. سپس از ایده‌ای همانند نمایش علمی اعداد استفاده می‌کنیم. هر چند خواهیم دید چون این اتفاق در فضای محدودی رخ می‌دهد دقت کار ما پایین خواهد بود.

در پایان با نمایش کاربردی BCD آشنا می‌شویم و عملیات پایه (جمع، تفریق، ضرب و تقسیم) آن را مورد بررسی قرار می‌دهیم.

جمع کننده‌ها:

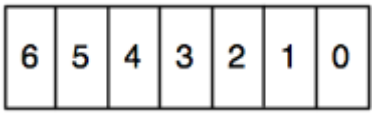
۲ عدد n بیتی داریم، ابتدا باید بدانیم این اعداد چه ساختاری دارند.



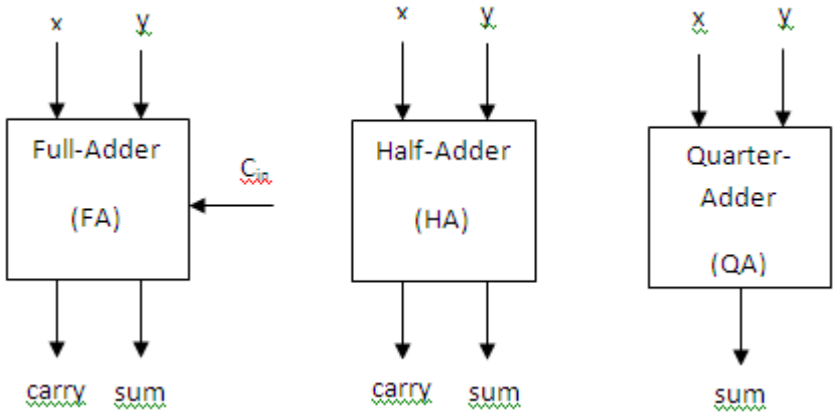
در روش اندازه علامت، علامت و عدد به صورت جدا از هم نگهداری می‌شوند. آخرین بیت (از سمت راست) نشان دهنده‌ی علامت است. در مکمل ۲ علامت و عدد در یک n بیت با هم نشان داده می‌شوند. در هر حال با وجود اعداد علامت دار دامنه‌ی اعداد قابل نمایش نصف می‌شود. چون هم اعداد مثبت داریم و هم اعداد منفی. در هر دو حالت علامت می‌تواند از طریق بیت آخر تشخیص داده شود. البته در روش مکمل ۲، برای پیدا کردن اندازه باید مکمل دوی عدد منفی را حساب کرد.

به طور کلی پیاده سازی اعمال حسابی نیازمند دانستن نوع عدد است و هر کدام نیز ملاحظات خاصی دارند.

برای سادگی فرض می‌کنیم اعداد صحیح و مثبت هستند (بدون علامت) بعدها تغییرات لازم برای هر نوع عدد را جداگانه بررسی خواهیم کرد.



در طراحی مدار معمولا اندیس گذاری از صفر و از سمت راست آغاز می‌شود.



:Half-Adder

X	Y	C	S
۰	۰	۰	۰
۰	۱	۰	۱
۱	۰	۰	۱
۱	۱	۱	۰

$s = sum = x \oplus y$

$c = carry = xy$

:Quarter Adder

X	Y	S
۰	۰	۰
۰	۱	۱
۱	۰	۱
۱	۱	۰

$s = sum = x \oplus y$

:Full-Adder

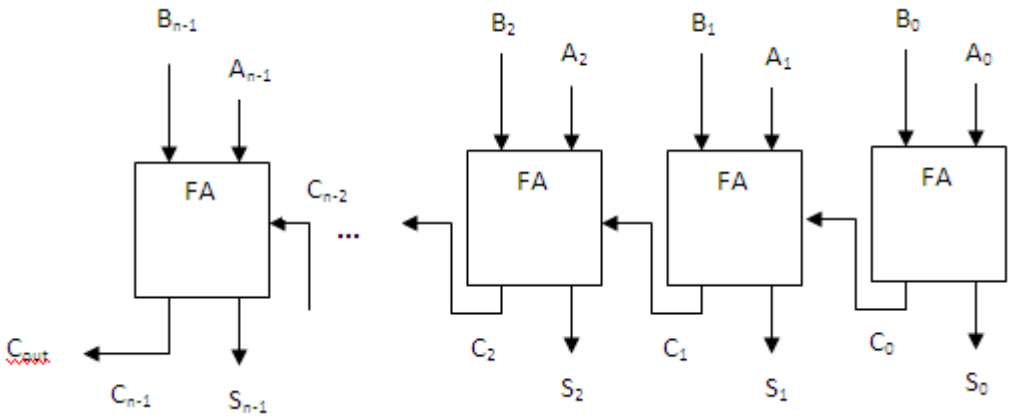
X	Y	C_{in}	C_{out}	S
۰	۰	۰	۰	۰
۰	۰	۱	۰	۱
۰	۱	۰	۰	۱
۰	۱	۱	۱	۰
۱	۰	۰	۰	۱
۱	۰	۱	۱	۰
۱	۱	۰	۱	۰
۱	۱	۱	۱	۱

$s = sum = x \oplus y \oplus C_{in}$

$C_{out} = carry = xy + C_{in}y + C_{in}x$

جمع کننده‌ی آبشاری (Ripple-Adder)

برای ساختن جمع کننده‌ی n بیتی یکی از روش‌های ساده استفاده از چند FA پشت سر هم است که در واقع همان تکنیک مورد استفاده‌ی انسان در محاسبات مبنای ۱۰ است.



در زمان ساخت ALU فاکتورهای مختلفی ظاهر می‌شوند که نقش اساسی در انتخاب نوع طراحی ما خواهند داشت. از این رو لازم است که هر کدام از اجزای مورد نیاز از نظر کیفیت و هزینه مورد سنجش قرار بگیرند.

در اینجا منظور از کیفیت، تاخیر در محاسبه پاسخ نهایی، و منظور از هزینه، تعداد گیت‌های لازم برای طراحی آن مدار است.

- HW cost: هزینه‌ی سخت افزاری- معمولا با تعداد ترانزیستورهای استفاده شده یا تعداد gate سنجیده می‌شود.
- Delay: به معنای زمان لازم برای دریافت خروجی از لحظه‌ی ورود داده است. برای ساده شدن محاسبات لازم برای بدست آوردن میزان تاخیر در مدار، میزان تاخیر هر گیت را یک مقدار ثابت فرض می‌کنیم.

در جمع کننده‌ی آبشاری داریم:

$$Cost(RippleCarry) = n * \text{Cost}(FA) = n[1(C_{xor}) + 3(C_{and}) + 1(C_{or})] = 5 * n(C_{gate}) = 5n$$

اما در مورد Delay، ابتدا لازم است ببینیم یک FA خود چه مقدار تاخیر دارد.

اگر فرض کنیم مقدار تاخیر ثابت گیت‌ها d است، خواهیم داشت

$$\begin{matrix}\text{Sum تاخیر} = d \\ \text{Cout تاخیر} = 2d\end{matrix} \Rightarrow \textit{Full Adder تاخیر} = 2d$$

در نتیجه برای جمع کننده‌ی آبشاری داریم:

$$Delay \begin{cases} Delay_{sum} = (n-1)2d + d = (2n-1)d \\ Delay_{Cout} = 2nd \end{cases}$$

جمع کننده‌ی آبشاری از ساده ترین نوع جمع کننده هاست. اما تـأخیر آن 2 nd اسـت و می‌تـوان بهتـر از ایـن طراحی انجام داد. از آنجایی که میزان تأخیر آن وابسته به تعداد بیت‌های ورودی است، با افزایش تعـداد بیت‌هـا شاهد رشد خطی زمان لازم برای محاسبه خروجی هستیم. به همین دلیل زمانی که می‌خواهیم هزینه‌ی کمتری بپردازیم ولی زمان مهم نیست از آن استفاده می‌کنیم. اما اگر زمان مهم باشد از جمع کننده‌های دیگـر اسـتفاده خواهیم کرد.

جمع کننده با پیش‌بینی بیت نقلی (Carry Look-ahead Adder (CLA)):

در محاسبه‌ی بیت‌های نقلی (Carry) داریم:

$C_0=A_0B_0+B_0C_{in}+A_0C_{in} \rightarrow C_0=A_0B_0+C_{in}(A_0+B_0)$

$C_1=A_1B_1+B_1C_0+A_1C_0 \rightarrow C_1=A_1B_1+C_0(A_1+B_1) \rightarrow C_1= A_1B_1+ (A_1+B_1)\, A_0B_0 + (A_1+B_1)\, A_0B_0C_{in}$

حال G و P را به صورت زیر تعریف می‌کنیم:

Generate $G_i=A_iB_i$

Propagate $P_i=A_i+B_i$

پس خواهیم داشت:

$C_0=G_0+C_{in}\, P_0$

$C_1=G_1+ G_0\, P_1+P_0\, P_1\, C_{in}$

$C_2=G_2+ G_1\, P_2+G_0P_1P_2 +P_0\, P_1P_2\, C_{in}$

$C_i= G_i+C_{i-1}\, P_i$

$C_{n-1}= G_{n-1}+G_{n-2}\, P_{n-1}+G_{n-3}P_{n-1}P_{n-2} +\ldots+P_0\, P_1\ldots P_{n-2}P_{n-1}\, C_{in}$

پس C_{n-1} یک SOP است که فقط G_iها و P_iها در آن ظاهر شده‌اند.

می‌توان G_iها و P_iها را پس ازتأخیر زمانی d به دست آورد. زیرا همزمان و به صورت موازی بیت‌های متناظر را and و or می‌کنیم. در مرحله‌ی بعد G_iها و P_iها را به یک مدار منطقی ترکیبی می‌دهیم و پس از 2d تأخیر، carryها را خواهیم داشت.

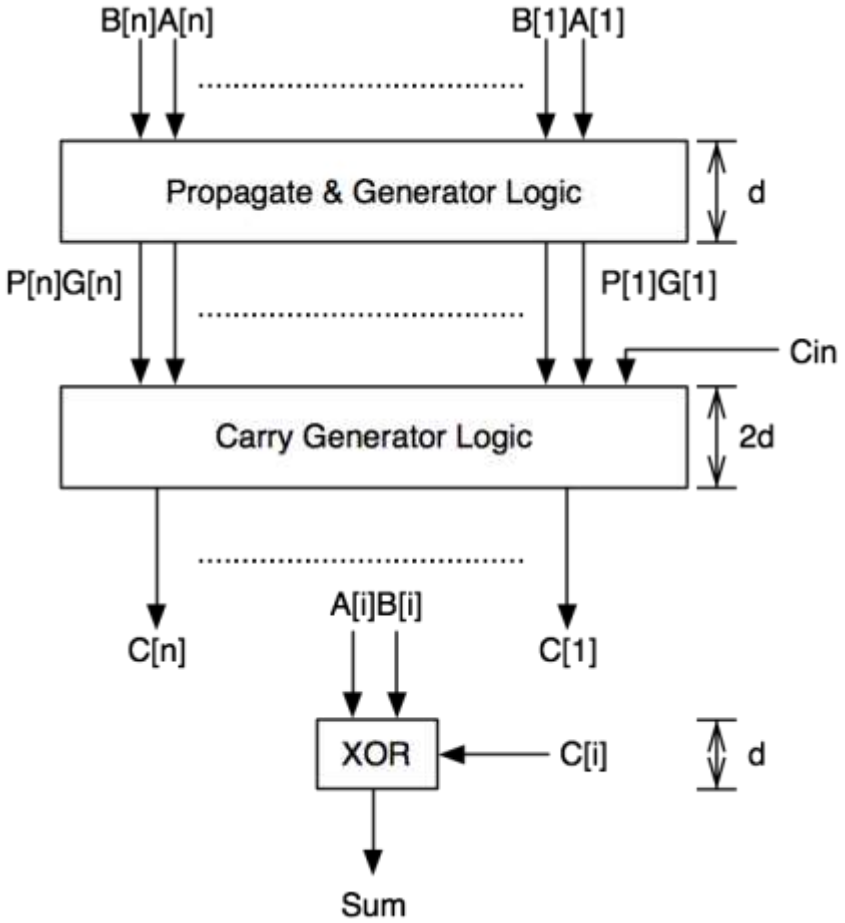
به این مدار ترکیبی که بیت‌ها را می‌گیرد و carry را به ما تحویل می‌دهد، Carry Generator و مدار جمع کننده‌ی حاصل از آن را Carry Look-ahead Adder می‌گویند.

با داشتن carryها و بیت‌های A و B، می‌توان با یک تمام جمع کننده (Full Adder) حاصل نهایی را حساب کرد. اما اینجا به C_{out} هم نیازی نداریم پس می‌توانیم از یک xor استفاده کنیم. این جمع کننده نیز پس از تأخیر زمانی d حاصل را محاسبه می‌کند (فقط یک گیت xor)، بنابراین در نهایت داریم:

$Delay_{sum}=4d$

$Delay_{carry}=3d$

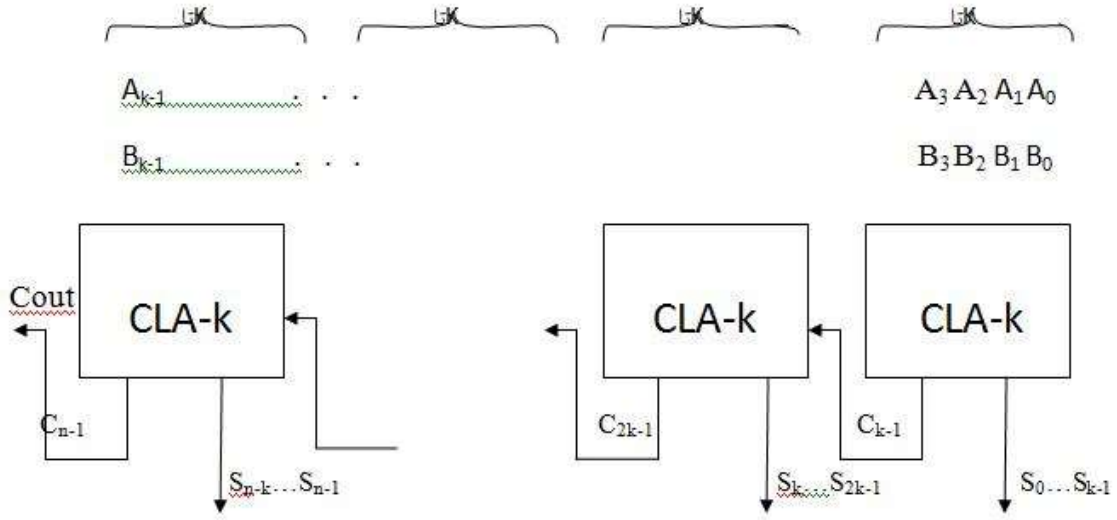
در زیر یک مدار جمع کننده با پیش بینی بیت نقلی را مشاهده می‌کنید.



طراحی Carry Look-ahead Adder تحول بسیار مهمی در جمع کننده‌ها به وجود آورد زیرا این اولین بار بـود که تأخیر در یک جمع کننده به تعداد بیت‌های ورودی وابسته نبود. این ابداع باعث شد که مرتبه‌ی زمانی جمع کننده‌ها از $O(n)$ به $O(1)$ کاهش یابد.

در Carry Look-ahead Adder (CLA) که طراحی کردیم هر and، حداکثر n ورودی خواهد داشت. در حـالی که چنین and هایی در عمل وجود ندارند. با بررسی ماکزیمم تعداد ورودی‌های گیت and در بازار می‌بینیم که حداکثر and ها ۴بیتی اند، پس می‌توان Carry Look-ahead Adder(CLA)های ۴ بیتی تولید کرد.

برای جمع اعداد n بیتی در عمل تعدادی Carry Look-ahead Adder (CLA)۴ بیتی را کنار هم می‌گذارند و مداری شبیه Ripple Carry Adder طراحی می‌کنند که البته سریع تر از Ripple Carry Adder ساده خواهد بود.



اگر فرض کنیم که این CLAهای k بیتی را بتوانیم طوری بسازیم که مثل قبل بیت نقلی را با 3d تأخیر و حاصل جمع را با 4d تأخیر به ما بدهد، خواهیم داشت:

$\text{Delay}_{\text{carry}} = 3d + 3d + \cdots + 3d = 3(\frac{n}{k})d$

$\text{Delay}_{\text{sum}} = (\frac{n}{k} - 1)3d + 4d = (3\frac{n}{k} + 1)\,d$

هرچه k کمتر باشد تأخیر بیشتر می‌شود. اگر k=n مانند همان n-bit-CLA عمل خواهد کرد و تأخیرها همان 3d و 4d خواهد شد. اگر k=1 مانند تمام جمع کننده‌ی عادی عمل می‌کند(و حتی بدتر از آن زیرا مدار در این حالت پیچیده تر شده و تاخیر افزایش می‌یابد). در حالت معمول k را ۴ قرار می‌دهند.

در این روش محاسبه‌ی G_iها و P_iها 2n گیت نیاز دارد و با محاسبه‌ی C_iها تعداد گیت‌ها از 5n بیشتر می‌شود و هزینه‌ی سخت افزاری هم بالا می‌رود اما در عوض کارایی تا حد خوبی افزایش و تأخیر کاهش می‌یابد.

در این مرحله از طراحی کارایی را به شکل زیر تعریف می‌کنیم:

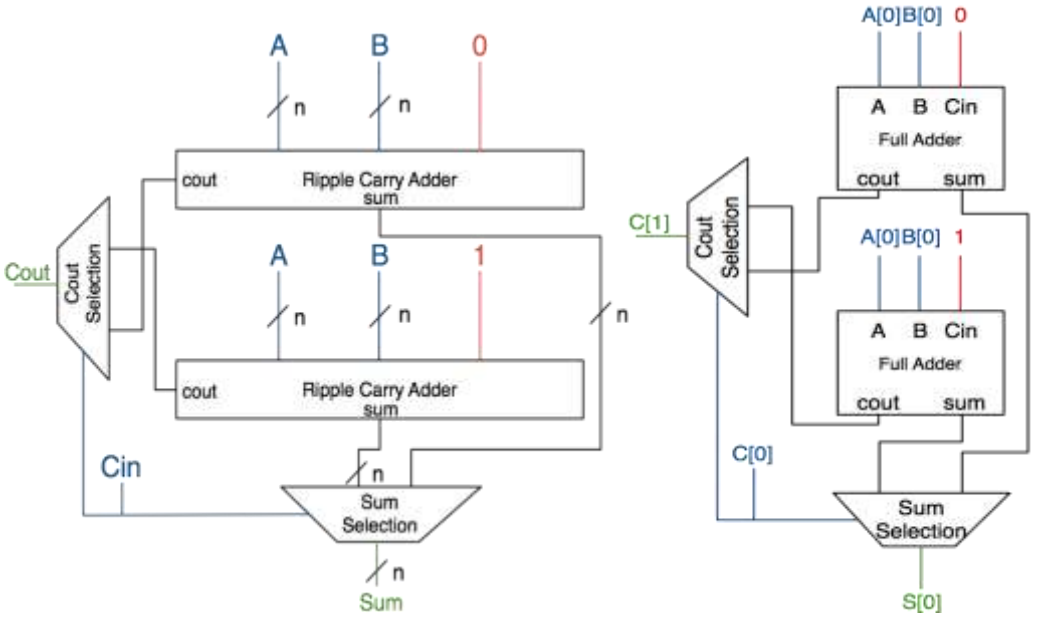
$$\text{performance} \sim \frac{1}{Delay * Cost}$$

مطابق با این تعریف، افزایش کارایی متناسب است با کاهش تاخیر و هزینه ساخت مدار.

جمع کننده‌ی انتخابی (Carry Select Adder):

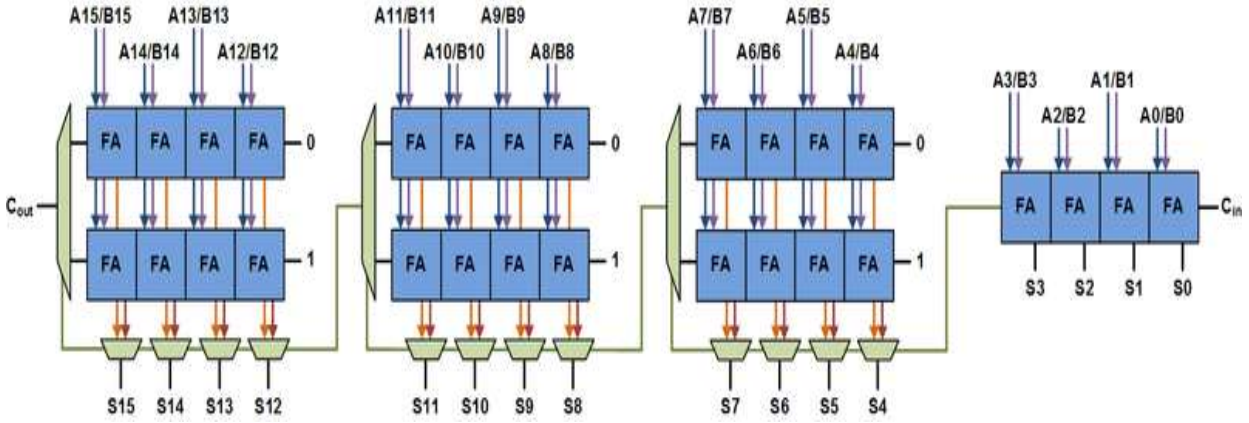
ایده‌ی اصلی طراحی CSA بر اساس این مشاهده است که در مقدار ورودی رقم نقلی فقط دو حالت صفر یا یک قرار خواهد گرفت. بنابراین زمان جمع کردن دو عدد n بیتی، کافی است به ازای هر ورودی یک بار جمع آبشاری با رقم نقلی ۰ و یک بار جمع آبشاری با رقم نقلی ۱ صورت گیرد. این دو محاسبه به طور موازی صورت می‌گیرند و پس از آن نتایج یکی از این دو جمع بر اساس رقم نقلی ورودی، به عنوان خروجی انتخاب می‌شود.

در زیر دو طراحی ساده بر اساس ایده‌ی CSA را مشاهده می‌کنید.



در شکل سمت چپ می‌توان به جای جمع کننده‌ی آبشاری از هر جمع کننده‌ی دیگری استفاده کرد. اما به هر حال با این شیوه طراحی تأخیر بیشتر از حالت ساده خواهد شد. (به دلیل وجود mux). علاوه بر این هزینه‌ی سخت افزاری هم افزایش پیدا کرده است. بنابراین طراحی ارائه شده هیچ مزیتی ندارد.

اما اگر به جای یک CSA از چند CSA که به هم به شکل آبشاری متصل شده اند استفاده کنیم، می‌توانیم تاخیر محاسبات را در شرایط خاص کاهش دهیم. به طور مثال اگر بلوک‌های ۴ بیتی CSA را به صورت آبشاری استفاده کنیم مدار زیر بدست می‌آید.



به این شیوه که از بلوک‌هایی با تعداد بیت مساوی در طراحی CSA استفاده شود، Uniform Carry Select Adder می‌گویند. تاخیر این نوع طراحی CSA به صورت زیر است:

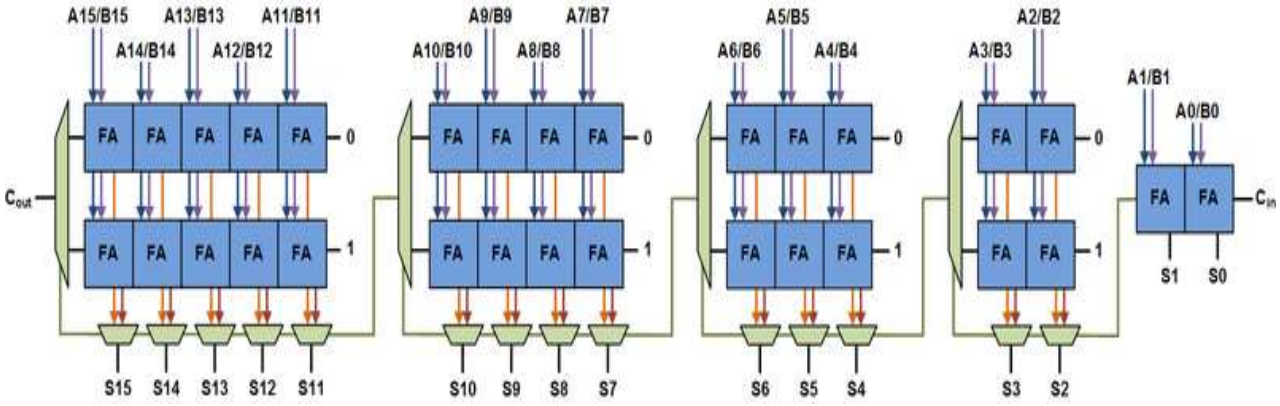
$$Delay = \left(\frac{n}{k}\right) 2d + 3kd$$

که k برابر است با تعداد سطح‌های مدار (در شکل بالا k=4). 3kd تاخیر به دلیل وجود mux هاست و همچنین 2d به دلیل تاخیر Full Adder های به کار رفته است. دقت کنید که انتخاب صحیح k می‌تواند شرایط مختلفی در تاخیر مدار ایجاد کند. به طور مثال اگر k=1، تاخیر برابر 2nd+3d می‌شود که از تاخیر جمع کننده‌آبشاری (2nd) بیشتر خواهد بود. اگر بخواهیم از جمع کننده آبشاری بهتر باشد خواهیم داشت:

$$\left(\frac{n}{k}\right) 2d + 3kd < 2nd \rightarrow 2nd + 3k^2d < 2nkd \rightarrow (-3)k^2 + (2n)k - 2n > 0$$

بنابراین کافی است که نامعادله‌ی فوق را برای بدست آوردن k مناسب حل کنیم، به طوری‌که تاخیر minimum شود.

از معایب این روش این است که بلوک (جمع کننده)های آخر مدت زیادی منتظر می‌مانند. یکی از کارهایی که برای افزایش کارایی می‌توان انجام داد، این است که تعداد بیت بیشتری برای جمع کردن به آن‌ها بدهیم. با همین ایده Non-Uniform Carry Select Adder طراحی شد. به این ترتیب جمع کننده‌ی بهتری خواهیم داشت.



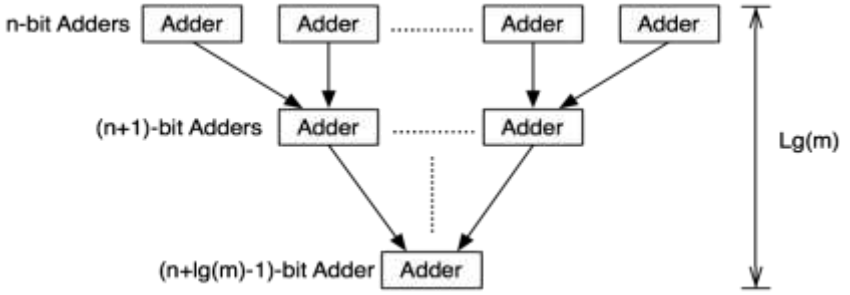
تأخیر در این روش کمتر می‌شود زیرا از تاخیر در مدارهای پایانی، برای جمع کردن ارقام بیشتر استفاده می‌شود و همچنین پهنای آن‌ها کوچکتر می‌شود و توان مصرفی نیز کاهش پیدا می‌کند. نسبت به طراحی قبلی (uniform carry select adder) کارآیی بهتری دارد ولی از نظر سخت افزاری تفاوت چندانی ندارند.

Carry Save Adder

فرض کنیم که قصد داریم m عدد n بیتی را جمع کنیم. اولین روشی که به ذهن می‌رسد این است که یک ماتریس m*n رقمی تشکیل دهیم و سطر به سطر جمع کنیم (با جمع کننده‌های آبشاری). در این روش حداقل تعداد جمع کننده‌های آبشاری m-1 تا خواهد بود (بین هر سطر) و هر کدام تأخیری برابر 2*nd خواهند داشت. بدین ترتیب تأخیر کل آن‌ها بسیار زیاد و به شکل زیر خواهد شد:

$$delay = (m-1) * 2nd = 2mnd$$

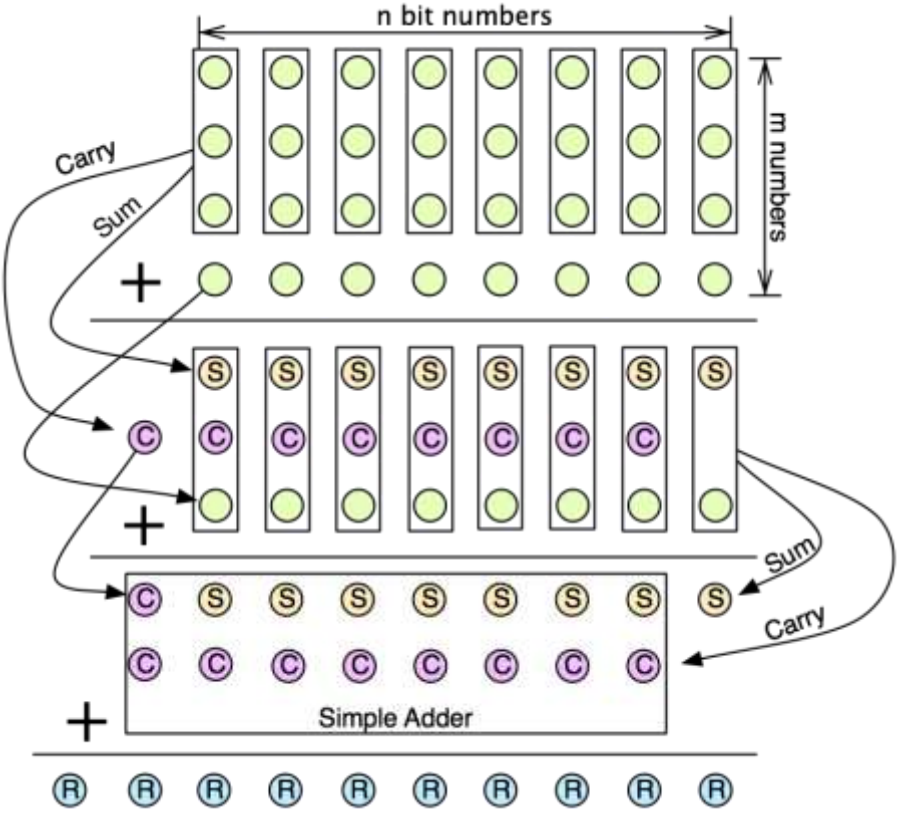
برای بهتر کردن این روش می‌توان جمع کننده‌ها را به صورت درختی قرار داد و به این شکل تعداد طبقات لازم را کاهش داد اما همچنان تأخیر زیادی خواهیم داشت.



$$delay = 2nd * [\log_2 m]$$

$$HW = m * 5ng$$

اما روش بهتر استفاده از جمع کننده‌ی دیگری به نام Carry Save Adder است. ابتدا بایستی به این نکته توجه کنید که یک Full Adder نقش یک جمع کننده‌ی سه تایی را ایفا می‌کند (دو ورودی عددی و یک cin). در این روش هر بار به کمک Full Adder ها دسته‌های سه تایی از اعداد را به دسته‌های دو تایی تبدیل می‌کنیم(C_{out} و حاصل جمع) و این عمل را تکرار می کنیم تا زمانی که به دسته های ۲تایی برسیم که در این مرحله با یک جمع کننده‌ی دیگر مانند جمع کننده‌ی آبشاری عملیات جمع به پایان می‌رسد.



ضرب کننده‌ها :

چنانچه دو عدد دودویی n بیتی A, B را در هم ضرب کنیم حاصل‌ضرب در حالت ماکزیمم 2n بیت خواهد شد. به عبارتی:

$$0 \leq A \leq 2^n$$

$$0 \leq B \leq 2^n$$

$$0 \leq A * B \leq 2^n * 2^n = 2^{2n}$$

یک ایده برای به دست آوردن نتیجه این است که حاصل‌ضرب هر بیت از B را در عدد دودویی A به طور جداگانه به دست بیاوریم و سپس نتیجه‌ی حاصل را با یکدیگر جمع کنیم. در مرحله iام اگر b_i صفر باشد، نتیجه حاصلضرب بیت iام B در عدد A برابر با صفر خواهد بود، در غیر این صورت چون بیت B یک است، نتیجه حاصلضرب این بیت در A برابر با خود A خواهد بود. (به هرکدام از این حاصل‌ضرب‌ها، حاصل‌ضرب‌های میانی یا بخشی گفته می‌شود.)

الگوریتم به این شیوه است که به ازای ضرب بیت i ام B در A، حاصلضرب میانی را i بیت به سمت چپ شیفت داده و در زیر حاصلضرب‌های میانی قبلی می‌نویسیم. بنابراین اولین حاصلضرب میانی، n بیت است با صفر بیت شیفت به سمت چپ. دومی n بیت است با یک بیت شیفت به چپ. .. . و nامین حاصلضرب میانی، به طول n بیت است با n بیت شیفت به چپ. لذا وقتی این حاصلضربهای میانی را باهم جمع کنیم، طول نتیجه نهایی به 2n بیت می‌رسد.

اگر چنین ایده ضربی را بخواهیم پیاده سازی نماییم، تعداد جمع کننده‌های مورد نیاز به شیوه سنتی، برابر با n-1خواهد بود. (یعنی برای جمع کردن n تا حاصلضرب میانی، به n-1 تا جمع کننده‌ی آبشاری n بیتی نیاز داریم.

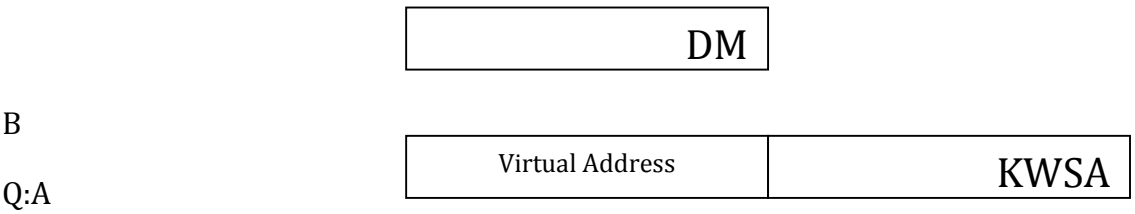
و چنانچه بخواهیم این ایده را با جمع کننده carry save adder پیاده سازی کنیم(در بخش قبل گفتیم که carry save adder برای جمع کردن m عدد n بیتی به کار برده می‌شود) باید carry save adderای برای جمع n عدد 2n بیتی بسازیم.

که پیاده سازی هر دو روش سخت است.

ضرب کننده ترتیبی:

لذا ایده ضرب کننده ترتیبی را مطرح می‌کنیم:

$$A \times B \rightarrow Q:A$$



۱) در این روش هدف، کم کردن پهنای جمع کننده هاست تا 2n بیتی نباشند.

۲) مقدار اولیه دو ثبات Q:Aدر الگوریتمی که در ادامه خواهد آمد، صفر است. (علامت: بین A,Q به این معناست که این دو ثبات به هم وصل یا به عبارتی concat شده‌اند و ۲n بیت جواب را تشکیل داده اند). و مقدار آن را هر بار با حاصلضرب‌های میانی جمع می‌کنیم. حاصلضرب نهایی در Q:A موجود خواهد بود.

۳) هر حاصلضرب میانی صفر است یا B. (در اینجا داریم B*A را حساب میکنیم) بنابراین در هر مرحله Q:A را یا با صفر جمع می‌زنیم یا با عدد B. خود عدد B هم n بیتی است؛ بنابراین یک جمع کننده n بیتی کافی است. چون هر کدام از این حاصلضرب‌های میانی بخواهد جمع شود در مرحله iام iتا شیفت به سمت چپ خورده است و اصلا با تعدادی از بیت‌ها در Q:A برای جمع در آن مرحله کاری نداریم. بنابراین لزومی ندارد که یک جمع کننده ۲n بیتی به کار ببریم.

به عبارت دیگر در هر مرحله باید اندیس i را نگاه کنیم و حاصلضرب میانی را که 0 یاB است به اندازه i تا به سمت چپ شیفت بدهیم و به ازای i+1 امین بیت A به بعد، عمل جمع را انجام دهیم. که در اینصورت به جمع کننده با پهنای بیتی بیش از n نیاز خواهیم داشت. بنابراین به جای چنین کاری، حاصلضرب میانی را ثابت نگاه می‌داریم و به جای شیفت دادن آنها به سمت چپ، Q:A را به سمت راست شیفت می‌دهیم. چون در مرحله i ام به i بیت سمت راست A که قبلا ازآنها برای تعیین حاصل ضربهای میانی استفاده کرده بودیم، نیازی نداریم. (به i بیت سمت راست A و n-i بیت سمت چپ Q در مرحله iام نیازی نداریم. بنابراین پهنای بیتی جمع کننده به nکاهش می‌یابد.)

با توجه به توضیحات فوق الگوریتم ضرب کننده ترتیبی به صورت زیر خواهد بود:

$n \rightarrow sc$

به A_0 نگاه کن:

اگر $A_0 = 0$؛ هیچ کاری انجام نده!

اگر $A_0 = 1: B + Q \rightarrow EQ$

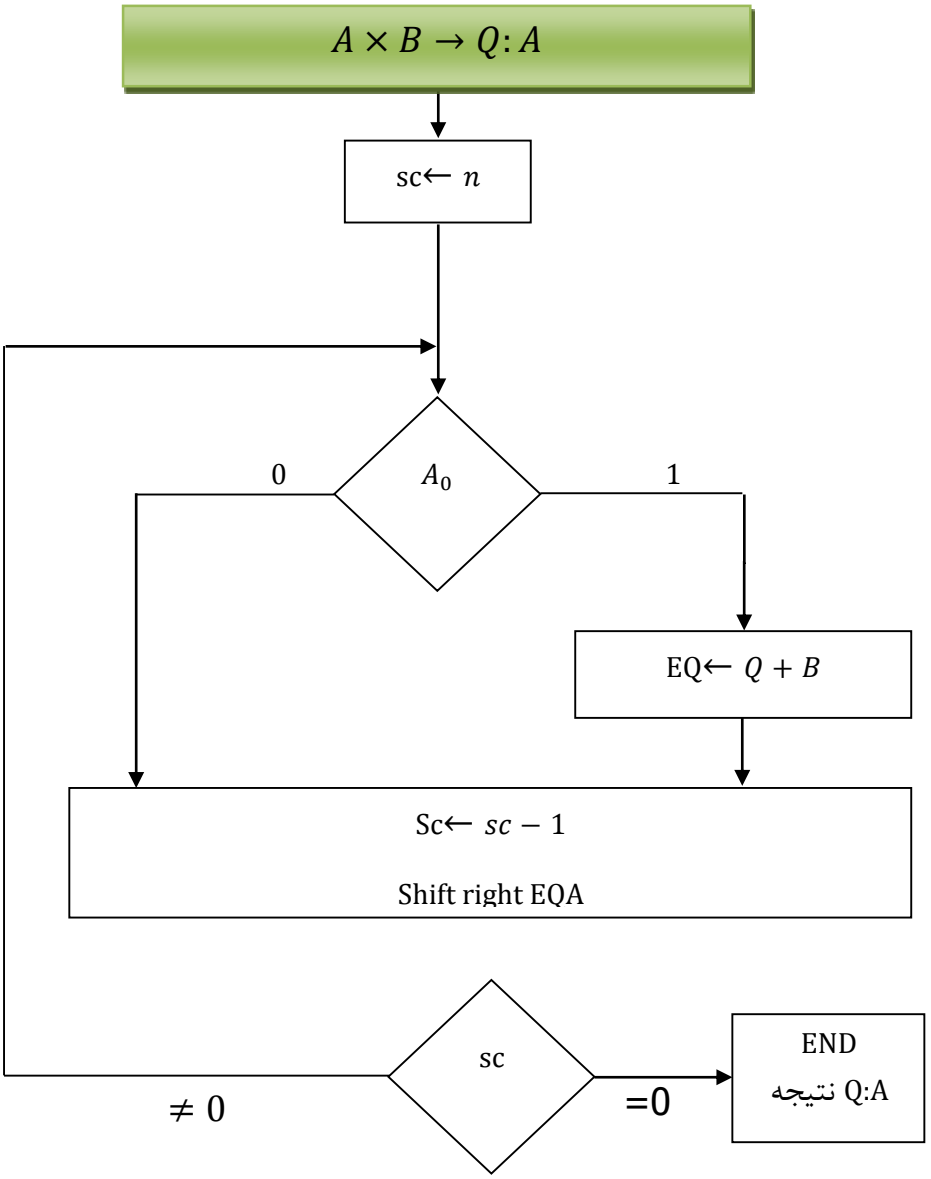
۱) E:Q:A را یک واحد به سمت راست شیفت بده!

۲) $sc - 1 \rightarrow sc$

۳) اگر sc=0؛ پایان؛ در غیر این صورت برو به مرحله ۲.

در این روش بدبینانه ترین حالت این است که هر n بیت A، یک باشد که در این صورت مدام باید عمل جمع را انجام بدهیم و تاخیر زیاد می‌شود.

خوش بینانه ترین حالت وقتی است که A تماما صفر باشد؛ در این صورت فقط n تا کلاک لازم است.

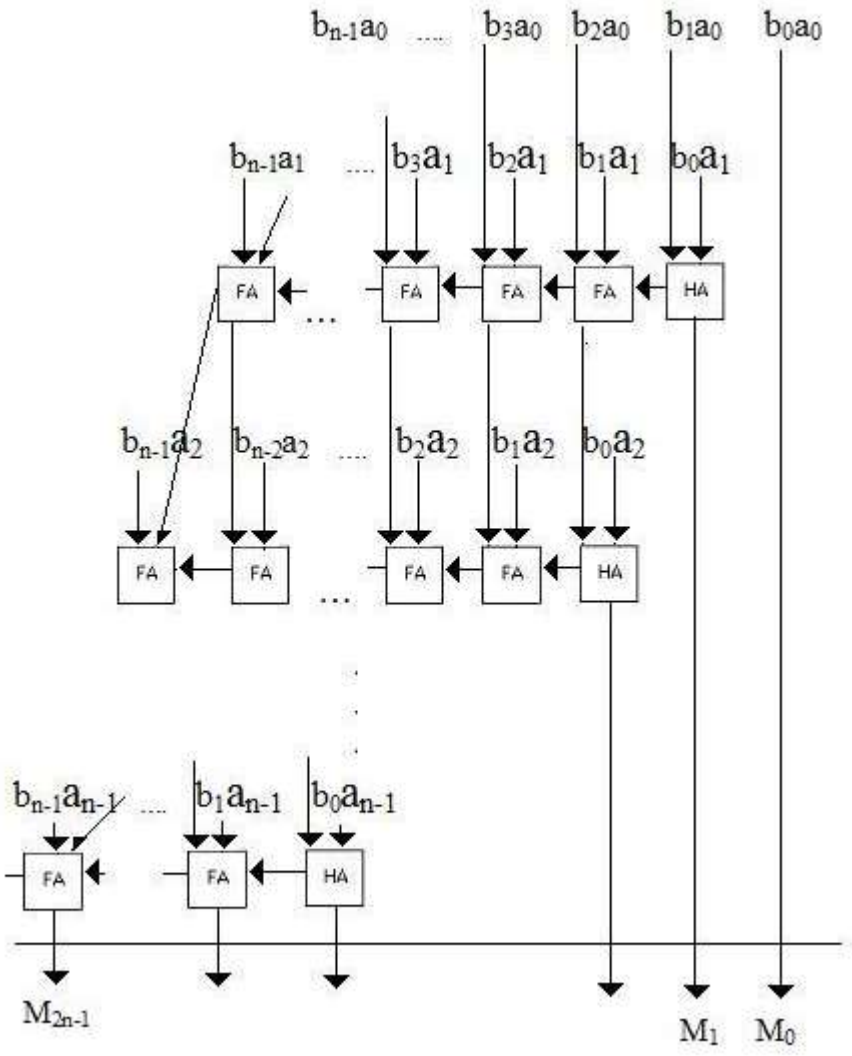


روش فوق یک روش خیلی ساده است که حداقل سخت افزار را مصرف می‌کند؛ چون فقط یک جمع کننده n بیتی و یک فلیپ فلاپ(E) و سه تا ثبات(Q,A,B) می‌خواهد.

اما از نظر زمانی چون این روش یه شمارنده میخواهد به n تا کلاک نیاز داریم و طول کلاک را هم سخت افزار جمع کننده می‌سازد

ضرب کننده آرایه ای

واحد ALU ذاتا ترکیبی است، پس ضرب را می‌توان ترکیبی هم پیاده سازی کرد که برای این مقصود از ضرب کننده آرایه‌ای استفاده می‌شود. برای ضرب دو عدد n بیتی a و b مانند شکل زیر عمل می‌کنیم:

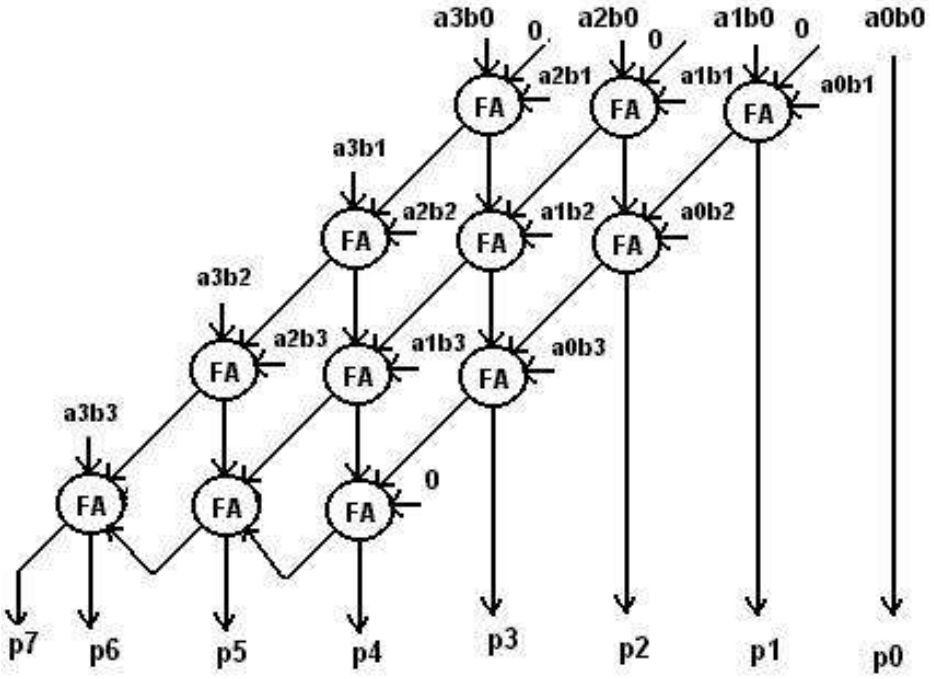


توضیح شکل:

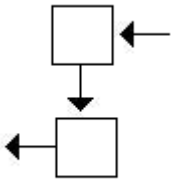
ردیف x ام شامل a_xb_j $(n \le j \le 0)$ها است. عدد هر ردیف را با عدد هم ستون ردیف پایینش جمع می‌کنیم و مجموع (sum) را به ردیف بعدی می‌دهیم تا چنانچه ردیف دیگری در پایین موجود بود با این حاصل جمع، جمع گردد و الا خود sum، M_j یعنی رقم j ام حاصل ضرب خواهد بود. همچنین در این جمع رقم نقلی (Carry out/Cout) ایجاد شده را به سمت چپ یعنی عدد هم ردیف در ستون بعد می‌دهیم تا به عنوان عدد نقلی

ورودی(Carry in/Cin) استفاده گردد. اگرتنها یک ردیف دیگر در ستونی که می‌خواهیم عدد جاری موجـود در آن را جمع کنیم وجـود داشـت بـه جـای تمـام جمـع کننـده(Full Adder/FA) از نـیم جمـع کننـده (Half Adder/HA) استفاده می‌کنیم.

در اینجا ذکر این نکته ضروری است که در ضرب کننده آرایه‌ای لزوما عدد نقلی به عـدد هـم ردیـف در سـتون بعدی داده نمی‌شود و گاهی به ردیف پایین تر در ستون بعد و. .. داده می‌شود. در هر حال نکته مهم ایـن اسـت که عدد نقلی ایجاد شده در هر ستون به نحوی باید در یکی از جمع‌های ستون بعد شرکت کند. نمونه دیگری از ضرب کننده آرایه‌ای مربوط به ضرب دو عدد ۴ بیتی را می‌توانید در شکل زیر مشاهده کنید:

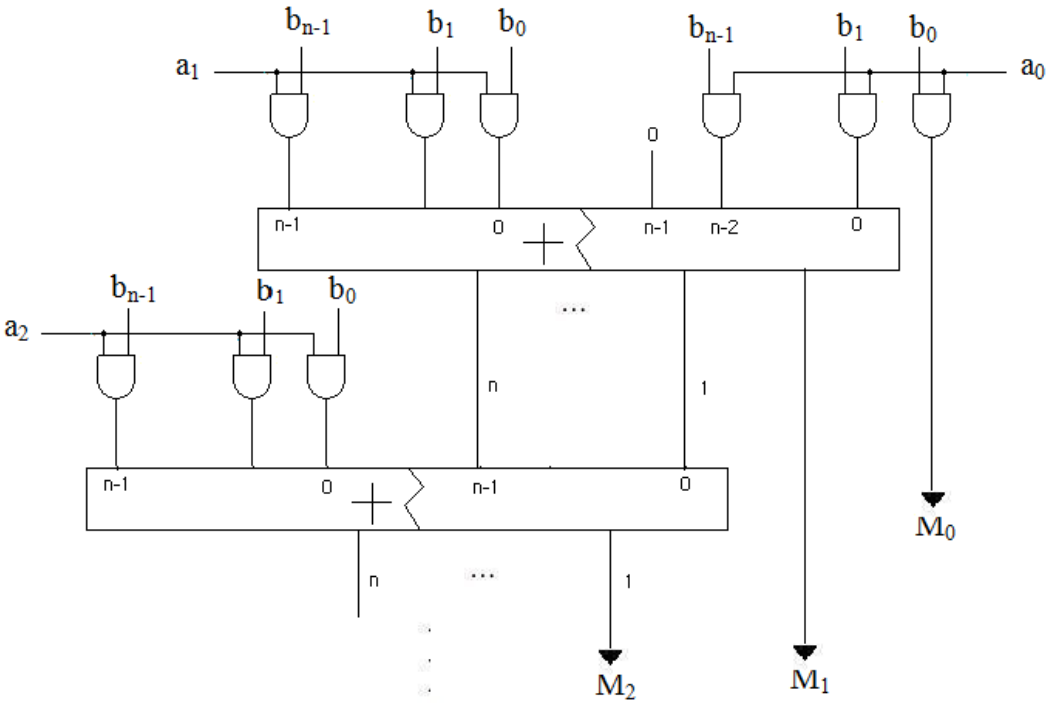


واضح است که در مدار ضرب کننده آرایه‌ای، سخت افزار زیادی مصرف می‌شود یعنی به تعـداد n-1، Cascade Adder. برای محاسبه تاخیر باید توجه داشت که برای تولید M_{2n-1} باید Carryها هم به صورت افقی و هـم بـه صورت عمودی (منظور sumهای تولید شده در هر FA است) حرکت کنند، یعنی:



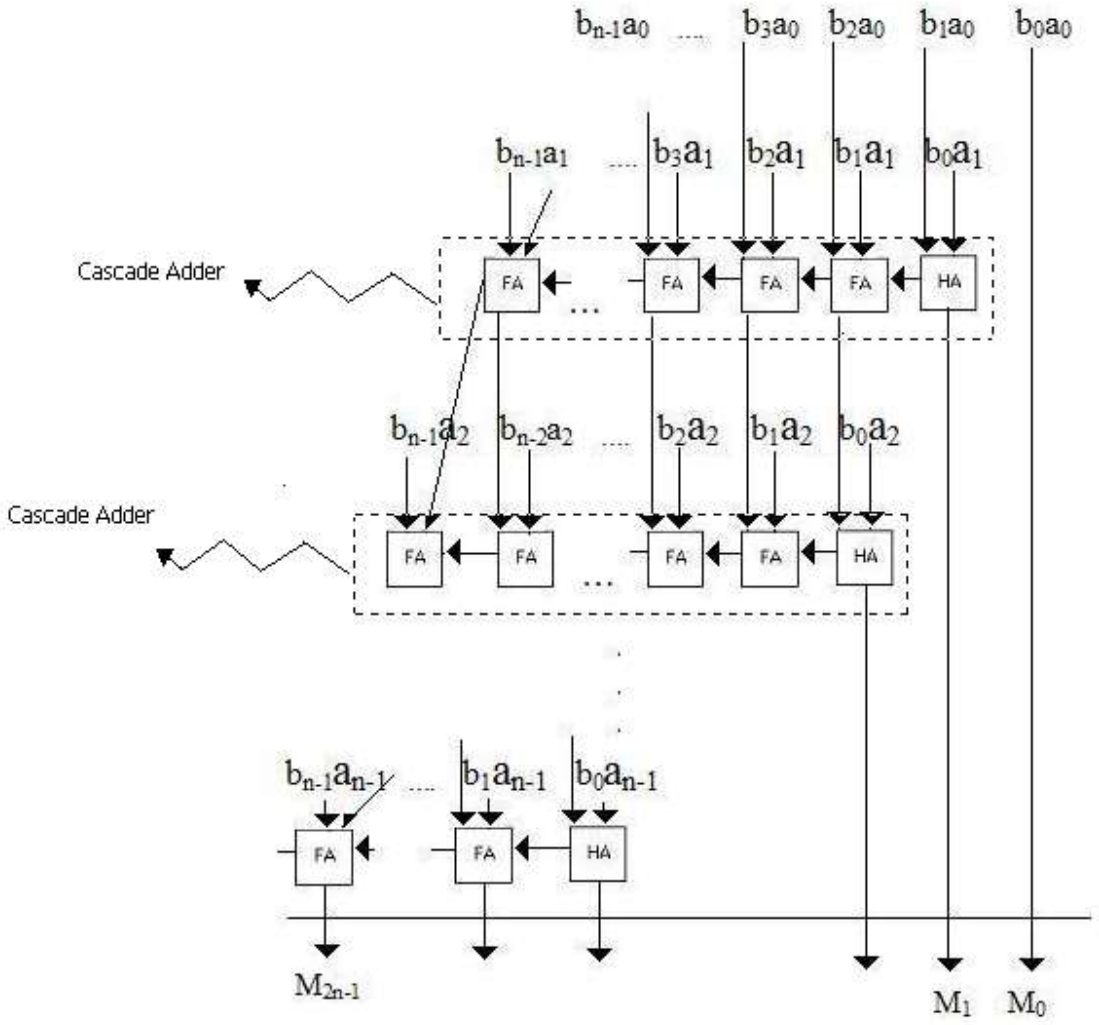
که تاخیر از مرتبه nd است در حالی که ضرب کننده ترتیبی(sequential) تاخیری از مرتبه n^2d داشت. (دقت کنیم که از آنجا که a_ib_jها موازی ایجاد می‌شوند پس تاخیر در تولید M_{2n-1} دقیقا برابر با 3nd+d است.)

نوع دیگری از ضرب کننده‌های آرایه ای:



در این نوع ضرب کننده آرایه‌ای هر بار M_k که راست ترین رقم حاصل ضرب است که تا کنون محاسبه نشده را برمی گردانیم. در این جا نیاز به n تا جمع کننده n بیتی داریم.

در واقع اگر جمع کننده‌ها را Cascade Adder انتخاب کنیم همان ضرب کننده آرایه‌ای قبلی را به ما می‌دهد، اما استفاده از ضرب کننده‌های دیگر نظیر CLA نوع متفاوتی از ضرب کننده آرایه‌ای را ایجاد می‌کند. برای روشن شدن موضوع به شکل زیر توجه کنید:



باید دقت داشت که ضرب کننده آرایه‌ای که در بالا مورد بحث قرار گرفت تنها برای اعداد مثبت استفاده می‌شود.

ضرب کننده بوث(Booth Algorithm/Multiplier)

در ضرب کننده ترتیبی، به تعداد ۱های موجود در مضروب، باید عملیات جمع روی مضروب فیه را انجام می‌دادیم. برای بهبود این الگوریتم و مستقل کردن ضرب از تعداد این ۱ها شخصی به نام بوث الگوریتم بوث را ارائه کرد که در این الگوریتم تعداد عملیات جمع و یا تفریق روی مضروب فیه تنها به تعداد ۰۱ و ۱۰هایی که در مضروب ظاهر می‌شوند بستگی دارد.

قبل از ذکر این الگوریتم ابتدا باید نمایش دیگری از اعداد را فرا بگیریم.

هر عدد در مبنای ۲ را می‌توان به صورت زیر در نظر گرفت:

...000011...111100.... . 000111.... 111...000

یعنی دنباله از ۱های متوالی و ۰های متوالی.

عدد x را در نظر بگیرید که :

عدد فوق در مبنای ۱۰ برابر است با $2^m - 2^l$

چرا که آن را می‌توان به صورت زیر نوشت:

$$\begin{array}{rl} & \dots 00100 \dots 0000 \dots \\ - & \dots 00000 \dots 0010 \dots \\ \hline x = & \cdots 00011 \dots 1110 \dots \end{array}$$

حال از آنجاییکه می‌توان هر عدد صحیح A را(به طور منحصر به فرد) به صورت مجموعه‌ای از ۱ها و ۰های متوالی یعنی به شکل حاصل جمع x_iها در نظر گرفت (x_iها ساختاری همانند x در بالا دارند یعنی تمام ۱ هایشان پشت سر هم آمده است) پس هر عدد صحیح را می‌توان به فرم $\sum 2^m - \sum 2^l$ نوشت.

با مثال‌هایی موضوع را روشن می‌کنیم:

مثال۱: $+5 = (00000101)_2 = 2^1\text{-}2^0\text{+}2^3\text{-}2^2$

مثال۲: $\text{-}10 = (11110110)_2 = 2^3\text{ -}2^1\text{-}2^4$

دقت شود که در مثال بالا رقم آخر یعنی سمت چپ ترین رقم ۱ است و تبدیل ۱ به ۰ رخ نمی‌دهد بنابرین نوشتن $+2^9$ درست نیست همان طور که در محاسبه نیز در نظر گرفته نشده است.

مثال۳: $(11000111)_2 = 2^3\text{-}2^0\text{-}2^6$

همان طور که پیشتر نیز گفته شد این روش تنها به مجموعه ۱های پشت سر هم وابسته است. در این روش بـه جای اینکه a_0 را نگاه کند دو بیت به دو بیت نگاه می‌کند و لبه‌های بالا رونده و پایین رونده مهمند. این دو رقم ۴ حالت دارد:

۰۰ ⇒ هیچ کاری نکن

۰۱ ⇒ عمل جمع را انجام بده

۱۰ ⇒ عمل جمع را انجام بده

۱۱ ⇒ هیچ کاری نکن

در نوشتن الگوریتم بوث از متغیرهای ۱ بیتی E و G و متغیر n بیتی Q استفاده می‌کنیم، که E، carry را شامل می‌شود و G نیز حاوی رقم اول از بین جفت ارقامی است که می‌خواهند بررسی شوند(A_0G بیان کننده یکی از ۴ حالت بالا است.). هدف ما ضرب دو عدد n رقمی A و B است ودر پایان عملیات ضرب Q خارج قسمت و A نیز باقی مانده خواهند بود.

الگوریتم بوث:

۱) $SC \leftarrow n$

۲) A_0G را نگاه کن اگر مساوی بود با:

۰۰: هیچ کاری نکن

۰۱: $EQA \leftarrow Q:A + B$

۱۰: $EQA \leftarrow Q:A - B$

۱۱: هیچ کاری نکن

۳) Shift Right (EQAG)

۴) $SC \leftarrow SC - 1$

۵) اگر $SC = 0$ آنگاه پایان؛ وگرنه برو به مرحله ۲

در این الگوریتم بهترین حالت این است که همه ارقام ۱۱ یا ۰۰ باشند. بدترین حالات نیز زمانی رخ می‌دهند که ارقام به طور متناوب از ۰ به ۱ و از ۱ به ۰ تغییر کنند:

۱) 1010.... 1010

در این حالت به تعداد $\frac{n}{2} - 1$ عمل جمع و $\frac{n}{2}$ عمل تفریق در الگوریتم بوث نیاز است.

۲) 0101.... 0101

در این حالت $\frac{n}{2}$ عمل جمع و $\frac{n}{2}$ عمل تفریق در الگوریتم بوث نیاز است.

تقسیم کننده:

گفتیم که اگر دو عدد n بیتی A,Bرا در هم ضرب کنیم، حاصل در حالت ماکزیمم در 2n بیت جا مـی‌گیـرد. در این قسمت می‌خواهیم تقسیم دو عدد را بررسی کنیم. چنانچه عدد 2nبیتی F را بر عدد nبیتی Bتقسیم نماییم حاصل باید درn بیت جا شود؛ البته لزوما تقسیم هر عدد 2n بیتی بر هر عدد nبیتی در n بیت جا نمی‌شـود. بـه عنوان مثال عدد 2n بیتی 1...11 را در نظر بگیرید. اگر این عدد را بر عدد n بیتی 1...00 تقسیم نماییم واضـح است که حاصل در n بیت نمی‌گنجد.

بنابراین ممکن است در برخی مواقع با سرریز مواجه شویم و سخت افزار نمـی‌توانـد محاسـبات را انجـام دهـد و عددی را که از طول استاندارد تجاوز می‌کند را نگه دارد. این حالات عبارتند از:

۱- اگر مقسوم علیه صفر باشد، حاصل بی نهایت می‌شود که نمی‌توان آن را در n بیت نمایش داد و سـرریز رخ می‌دهد.

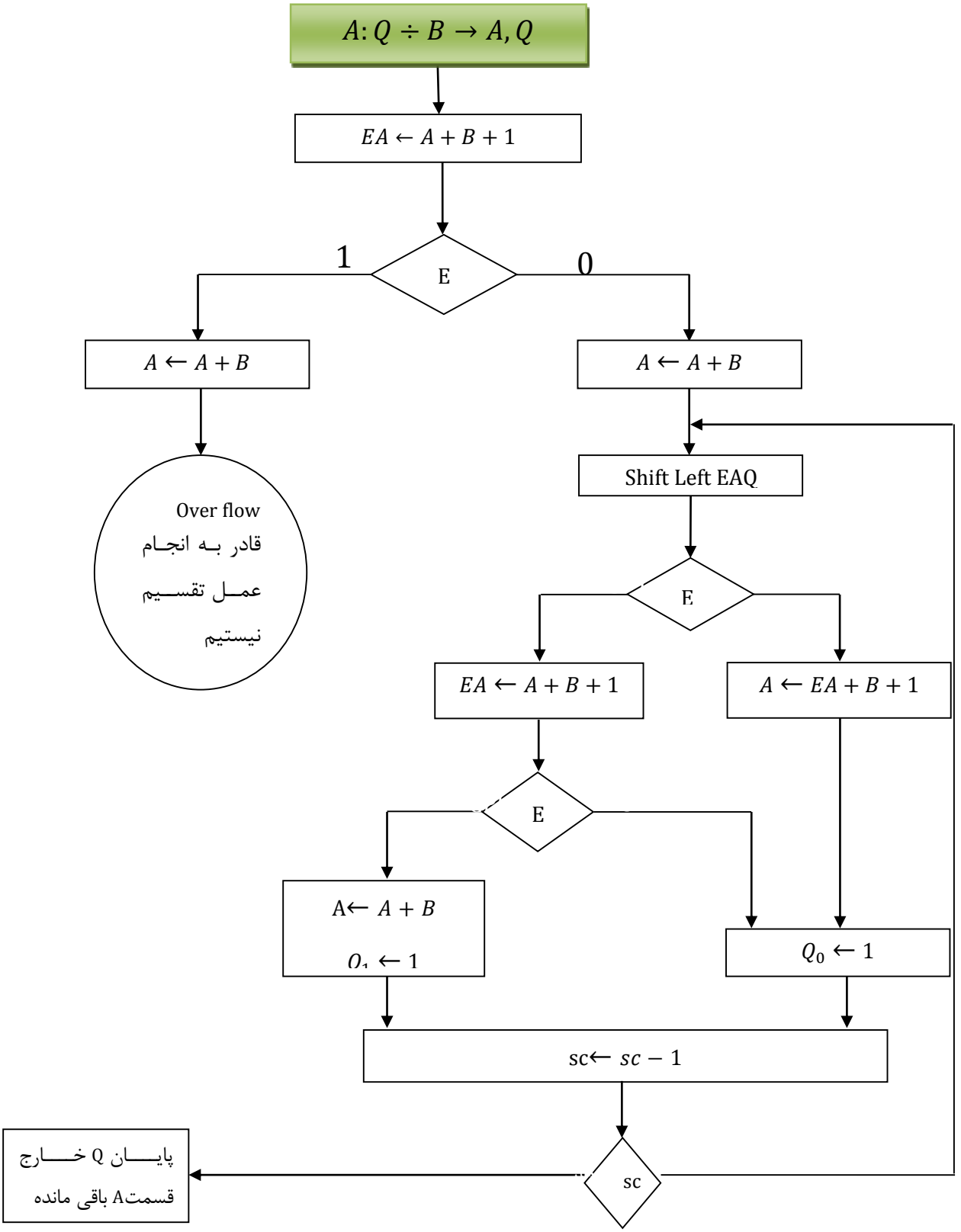
۲- اگر خارج قسمت در n بیت جا نشود، نیز سـرریز رخ مـی‌دهـد. در واقـع چنانچـه عـدد ۲n بیتـی $F =$ $F_H: F_1$ بر عدد n بیتی Bتقسیم شود، اگر $F_H \geq B$، آنگاه خارج قسمت در nبیت جا نمی‌شود.

اگر کمی دقت کنید متوجه می‌شوید که حالت یک، در واقع زیر مجموعه‌ای از حالت ۲ اسـت. بنـابراین سـاختن سخت افزار حالت ۲ کافی است.

حالت سرریز معمولا با یک شدن فلیپ فلاپ خاصی که به آن فلیپ فلاپ سرریز گوییم مشخص می‌شود.

معمولا فرضمان این است که اعداد مثبت هستند.

فلوچارت الگوریتم تقسیم به صورت زیر است:



در این روش از ۴ ثبات، یک فلیپ فلاپ و یک جمع کننده استفاده کردیم. یک ثبات برای نگهداری مقسوم علیه و دو ثبات برای نگهداری مقسوم.

در این روش ابتدا باید چک کنیم که در تقسیم با سرریز مواجه خواهیم شدیا نه؟ به این منظور مقسوم علیه(B) را از بیت‌های نیمه باارزش تر مقسوم (A)کم میکنیم و در EA قرار می‌دهیم. چنانچه در این عمل carry مساوی یک باشد، همانگونه که در جلسات قبلی ثابت شد A ازB بزرگتر بوده است؛ لذا با حالتی مواجه هستیم که سرریز رخ میدهد و باید فلیپ فلاپ سرریز را یک کنیم. همچنین بیتهای نیمه پرارزش مقسوم را که در هنگام تفریق خراب نموده ایم باید به حالت قبل برگردانیم. لذا Bرا با A جمع می‌کنیم که A قبلی حاصل شود و آن را در A میریزیم.

اما چنانچه carry حاصل صفر باشد، یعنی تقسیم با سرریز مواجه نمی‌شود. باز هم A را به حالت قبلی خود برمی گردانیم و از آنجایی که مشخص شد که در بار اول n بیت سمت چپ مقسوم از عددn بیتی مقسوم علیه کوچکتر بوده، پس رقم اول خارج قسمت صفر است. لذا EAQ را یک بیت به سمت چپ شیفت میدهیم. حالا یک بیت از Q آزاد می‌شود که می‌توانیم در مرحله بعد برای نگهداری خارج قسمت از آن استفاده نماییم.

ما پیش‌فرض، بیت خارج قسمت در هر مرحله را برابر با یک می‌گیریم. چنانچه نادرست بود، آن را به صفر تغییر می‌دهیم. برای تشخیص درستی یا نادرستی این فرض مقسوم علیه را از A کم می‌کنیم. طبق همان قضیه‌ای که قبلا گفتیم جنانچه carry حاصل یک باشد، یعنی Aاز مقسوم علیه بزرگتر بوده، لذا خارج قسمت یک درست بوده است و بیت صفرام Q را یک میکنیم. همچنین باقیمانده جزئی در این مرحله هم همان حاصل تفریق B از A است که در Aگذاشته شده است. اما چنانچه carry حاصل، صفر باشد، به این معنی است که A از مقسوم علیه کوچکتر بوده است. واضح است که در این حالت، خارج قسمت در این مرحله صفر است. پس علاوه براین که باید بیت صفرام Q را صفر کنیم، باید Aرا هم به همان حالت قبل از عمل تفریق برگردانیم. چون وقتی خارج قسمت صفر باشد، باقیمانده جزیی برابر با با همان A خواهد بود.

حال از شمارنده یکی کم می‌کنیم. چنانچه شمارنده به صفر رسید عملیات تقسیم به پایان رسیده است؛ در غیر این صورت برای ادامه عمل تقسیم به مرحله شیفت دادن به چپ بازمی گردیم و عملیات را ادامه می‌دهیم.
در پایان Qخارج قسمت و A باقی مانده نهایی است.

گفتیم در این الگوریتم فرض می‌شود که یک عدد مثبت 2n بیتی به یک عدد مثبت n بیتی تقسیم می‌شود. حالا اگرشیوه نمایش عدد اندازه- علامت یا مکمل دو باشد ابتدا باید تکلیف آن را مشخص نمود. به این صورت

که اگر اعداد مثبت باشند که همین روند را انجام می‌دهیم. اما اگر یک یا هردو اعداد منفی باشند ابتدا معادل مثبت آن را به دست می‌آوریم؛ سپس عمل تقسیم را برای آن دو انجام می‌دهیم و در نهایت تعیین علامت می‌کنیم.

شیوه نمایش	جمع و تفریق	ضرب و تقسیم
مکمل دو	ساده	پیچیده
اندازه-علامت	پیچیده	ساده

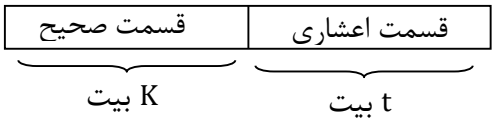
اعداد اعشاری

دو نوع دسته‌بندی برای اعداد اعشاری وجود دارد:

- ممیز ثابت
- ممیز شناور

ممیز ثابت(Fixed point)

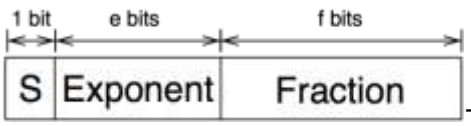
در اعداد اعشاری با ممیز ثابت, تعداد بیت‌های اختصاص داده شده به قسمت صحیح و اعشاری ثابت می‌باشد، در زیر قالب این نوع عدد اعشاری را مشاهده می‌کنید:



همانطور که مشخص است, در این حالت, هر دو قسمت با هم عدد را تشکیل می‌دهد. در قسمت اول، تکه‌ی صحیح به صورت مکمل ۲ ذخیره می‌شود، قسمت اعشاری نیز به شکل بدون علامت در t بیت آخر ذخیره می‌شود. همانطور که مشاهده می‌کنید قسمت‌های اختصاص داده شده برای اعداد اعشاری همواره ثابت است.
محاسبات ممیز ثابت از نظر پیاده سازی بسیار ساده است.

ممیز شناور(Floating point)

در اعداد ممیز شناور، نگهداری اعداد اعشاری به شکل متفاوتی انجام می‌شود. ایده‌ی اصلی این روش نگه‌داری اعداد در حافظه به شکل علمی آن‌هاست. قالب پیشنهادی مورد استفاده برای ذخیره سازی اعداد به این روش در شکل زیر قابل مشاهده است.



$\rightarrow Floating\ point\ number = (-1)^S * 1.F * 2^E$

در این نحوه‌ی ذخیره سازی اعداد، S بیانگر علامت عدد (مثبت ۰، منفی ۱)، Exponent بیانگر نمای توانی است که به صورت مکمل ۲ ذخیره می‌شود. Fraction یا مانتیس بیان‌گر قسمت اعشاری عدد نرمال(هنجار) شده است. (عدد نرمال شده در نمایش علمی، عددی است که در قسمت صحیح، فقط یک رقم وجود داشته باشد) با توجه به اینکه در مبنای ۲ هستیم، این تک رقم صحیح به غیر از زمانی که مقدار عددی صفر باشد همواره ۱ خواهد بود. بنابراین در این حالت کافی است که فقط قسمت اعشاری در قسمت Fraction ذخیره شود.

حال این سوال مطرح می‌شود که چه طور می‌توان یک عدد را به صورت ممیزشناور نمایش داد. برای این کار لازم است عملیات هنجارسازی صورت بگیرد. هنجارسازی همان تبدیل نمایش عدد به صورت علمی است. به طور مثال:

نمایش هنجار سازی شده	نمایش ناهنجار
$\mathbf{1.1111 * 2^4}$	11111
$\mathbf{1.111 * 2^{-3}}$	0.001111
$\mathbf{-1.000001 * 2^2}$	-100.0001

در صورتی که فضای ذخیره سازی عدد ۱۰ بیت باشد و ۴ بیت برای توان و ۵ بیت برای مانتیس، به ترتیب خواهیم داشت:

S(1bit)	Exponent (4bits)	Fractions (5bits)
0	0100	**11110**
0	1101	**11100**
1	0010	**00000**

دقت کنید که در قسمت Fraction دقیقا ۵ رقم اول اعشار به همان شکل در حافظه ذخیره شده است.

در طی تبدیل به اعداد ممیز شناور همواره باید به نکات زیر توجه داشت:

۱. همه‌ی اعداد به جز 0 قابلیت هنجار شدن دارند. برای همین 0 را با کوچک ترین عددی که می‌توان نشان داد، نمایش می‌دهیم. (در قراردادهای خاص برای بیان صفر از نمایش متفاوتی استفاده می‌شود.)

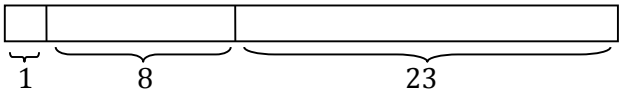
۲. قسمت اعشاری(=mantis) همیشه مثبت است و دقیقا به همان شکل در حافظه نگهداری می‌شود.

۳. قسمت نما(=exponent) می‌تواند مثبت یا منفی باشد، که به صورت مکمل‌دو نشان داده می‌شود.

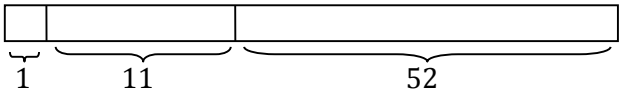
۴. قسمت‌های f و e در سیستم نمایش اعشاری چند بیت هستند؟ چون محدودیت ذکر شده تاثیر زیادی در حدود قابل ذخیره سازی توسط سیستم ایفا می‌کند. همواره لازم است که بدانیم آیا عدد محاسبه شده در حافظه قابل ذخیره سازی است یا نه. این عدد می‌تواند برای نمایش بسیار بزرگ باشد(سر ریز در ممیز شناور: زمانی اتفاق می‌افتد که توان مثبت آنقدر بزرگ باشد که در محدوده‌ی توان جا نشود)، همچنین این عدد می‌تواند بسیار کوچک باشد(زیرریز در ممیز شناور: زمانی اتفاق می‌افتد که توان منفی آنقدر بزرگ باشد که در محدوده‌ی توان جا نگیرد). در صورت وقوع هر کدام از این اتفاق‌ها سیستم پاسخ غلط خواهد داد.

مطابق با استاندارد IEEE754 دو پیشنهاد برای تنظیم فضای f و e ارائه شده است.

1. Single Precision(دقت ساده): 32 bits (Exponent 8 bits, Fraction 23 bits)



2. Double Precision(دقت مضاعف):64 bits (Exponent 11 bits, Fraction 52 bits)



زمانی که لازم است دو عدد اعشاری را با یکدیگر مقایسه کنیم، باستی ابتدا به توان آن توجه کنیم. با توجه به اینکه عددهای توان در قالب مکمل ۲ ذخیره می‌شوند و می‌توانند منفی باشند، عملیات مقایسه دشوار می‌شود. برای ساده تر کردن عملیات مقایسه، تصمیم بر این شد که نماها را به مقدار ثابتی در فضای عددی جابه‌جا کنیم. به این منظور، توان هر عدد را با عدد 2^{e-1} جمع می‌کنیم تا توان تمامی اعداد مثبت شود. امر سبب می‌شود که مقایسه اعداد اعشاری با ممیز شناور، ساده‌تر گردد. در جدول زیر تاثیر این عملیات نشان داده شده است:

4 + مکمل‌دو	مکمل دو	عدد
111	011	۳
110	010	۲
101	001	۱
100	000	۰
011	111	-۱
010	110	-۲
001	101	-۳
000	100	-۴

دقت کنید که با اعمال این روش (که به bias کردن شناخته می‌شود)، کوچکترین عدد (۴-) به مقدار صفر رسید و بزرگترین عدد به مقدار ۷ (یا ۱- در نمایش مکمل ۲). حال اگر اعداد را به صورت بدون علامت فرض کنیم، مقایسه‌ی آن‌ها بسیار ساده تر خواهد بود. در اثر این تبدیل، ترتیب اعداد ثابت باقی مانده است و مقایسه‌ی آنها نیز ساده تر شده.

با توجه به اینکه بعضی از اعداد اصم کاربرد زیادی دارند (مانند e و π و $\sqrt{2}$ و ...). بهتر است که آن‌ها در قالب ویژه‌ای ذخیره کنیم تا دقت محاسبات اعشاری خود را افزایش دهیم. به همین منظور نما را با عددی مثل $2^{e-1} - x$ جمع می‌کنیم و x ردیف از توان را به اعداد خاص اختصاص می‌دهیم. بر خلاف روش نشان داده شده، در این حالت کوچکترین عدد در توان صفر نخواهد شد. به طور مثال اگر x=1 باشد، عدد ۳- به صفر تبدیل می‌شود و عدد ۴- مقدار ۱۱۱ خواهد گرفت. حال حالتی که توان ۱۱۱ باشد را حالت خاص فرض می‌کنیم و از ردیف آن، برای ذخیره سازی اعداد خاص استفاده می‌کنیم. به این شکل بازه‌ی مثبت و منفی توان که قبلا از ۳ تا ۴- بود، متوازن شده و به ۳ تا ۳- تبدیل می‌شود و مقدار عددی ۴- که به شکل ۱۱۱ ظاهر می‌شود برای

نگهداری اعداد خاص استفاده خواهد شد. معمولا اعداد خاص در قالب کد و در قسمت Fraction عـددی ظـاهر می‌شوند که توان آن یکی از این توان‌های اختصاص یافته است. برای فهمیدن مفهوم این کدها بایستی از جدول مخصوص آن استفاده کنیم.

یکی دیگر از حالات خاص ممکن، مقدار Not a Number = NaN (ناعددی)است. زمانی که نتیجه‌ی محاسـبات عدد نباشد، باید به شکلی قابل ذخیره کردن باشد، از این رو باید برای این حالت خاص نیـز یـک شـکل در نظـر گرفت. به طور مثال زمانی که از یک عدد منفی جذر می‌گیرید، خروجی مقدار عددی نخواهد بود و باید حاصـل در این قالب ذخیره شود.

طبق قرار داد $\text{bias \#1}= 2^{e-1}$ و $\text{bias \#2}= 2^{e-1}-1$ تعریف کرده و به نمایشی که مقدار توان را با bias #1 و یا bias #2 جمع کند، به ترتیب biased #1 و biased #2 گفته می‌شود. دقت شود کـه مقـدار bias لزومـا مقادیر یاد شده نیست و ممکن است در شرایط مختلف مقادیر مختلفی باشد که به تعریف بستگی خواهد داشت. به طور مثال اگر ۳ بیت برای e در نظر بگیریم، مطابق با روش bias #1 باید به توان هر عدد مقـدار ۴ را اضـافه کنیم. به شکل مشابه ممکن است که تعریف شود مقدار bias برابر با ۴ است که هر دو یک معنی را می‌رسـاند و آن هم این است که باید توان را با ۴ جمع کنید.

با توجه به اینکه تعداد اعداد اعشاری ممکن حتی در بازه‌ی ۰ تا ۱ بی‌نهایت است، مسلما قادر به ذخیـره سـازی تمامی آن‌ها نخواهیم بود، از این رو فاکتورهای مورد توجه در توصیف نمایش اعداد ممیز شناور شامل ۴ دسته‌ی کلی می‌شود.

۱- چه تعداد عدد در این روش قابل ذخیره‌سازی است؟

۲- بازه‌ی اعداد قابل ذخیره سازی چقدر است؟

۳- دقت اعداد ذخیره شده چقدر است؟

۴- حداقل و حداکثر عدد قابل ذخیره سازی چقدر است؟

در ادامه با بررسی دقیق مقادیر قابل ذخیره سازی در اعداد شناور تلاش می‌کنیم که به سـوال‌هـای فـوق پاسـخ دهیم.

در مرحله‌ی اول لازم است که به چند سوال پاسخ دهیم، برای ادامه‌ی بحث فرض کنید که طول قسمت Fraction در عدد ممیز شناور f بیت، و همچنین طول قسمت Exponent در عدد ممیز شناور ما e بیت باشد.

۱- حداقل مقدار Fraction چیست؟ با توجه به بدون علامت بودن قسمت Fraction کمترین مقدار آن برابر با صفر است. (تمامی ارقام صفر)

۲- حداکثر مقدار Fraction چیست؟ با توجه به بدون علامت بودن قسمت Fraction بیشترین مقدار آن برابر است با $1 - 2^{-f}$ (دقت کنید که مقدار Fraction در قسمت اعشاری عدد است، و زمانی که تمام مقادیر آن ۱ شود، حاصل برابر خواهد بود با $1 - 2^{-f}$)

۳- حداقل مقدار Exponent چیست؟ با توجه به علامت دار بودن عدد، حداقل مقدار Exponent برابر است با -2^{e-1}

۴- حداکثر مقدار Exponent چیست؟ با توجه به علامت دارن بودن عدد، حداکثر مقدار Exponent برابر است با $2^{e-1} - 1$

با توجه به پاسخ سوالهای فوق، می‌توانیم کمترین مقدار مثبت قابل ذخیره در عدد ممیز شناور را بدست آوریم. بدیهی است که مطابق با قالب عدد ممیز شناور، کمترین مقدار مثبت ممکن برای آن به شکل زیر قابل محاسبه است:

$$\varepsilon = N_{min} = 1.F_{min} * 2^{E_{min}} \rightarrow 1.0 * 2^{-2^{e-1}}$$

این عدد به اسم اپسیلون شناخته می‌شود و در بعضی مواقع معادل صفر فرض می‌شود. در صورتی که e برابر با ۸ بیت باشد، مقدار اپسیلون برابر است با 2^{-128} که مقدار خیلی کوچکی است. در صورتی که از نمایش از پیش تعریف شده‌ی biased #1 تبعیت کنیم، تمامی مقادیر ذخیره شده برای این عدد ۰ خواهد بود (چرا؟).

کوچکترین عدد قابل ذخیره سازی، بعد از اپسیلون چیست؟ قاعدتا بایستی مقدار Fraction را افزایش دهیم (چرا؟). کمترین مقداری که می‌توان Fraction را افزایش داد برابر است با 2^{-f} . پس برای عدد بعدی خواهیم داشت.

$$N_{min+1} = \left(1 + 2^{-f}\right) * 2^{-2^{e-1}}$$

دقت کنید که اختلاف این عدد با عدد قبلی برابر است با $\Delta_1 = 2^{-f} * 2^{-2^{e-1}}$ و به همین ترتیب مقادیر افزایش پیدا می‌کنند تا زمانی که به بیشترین مقدار Fraction برسیم.

$$N_{min+2^f-1} = \left(2 - 2^{-f}\right) * 2^{-2^{e-1}}$$

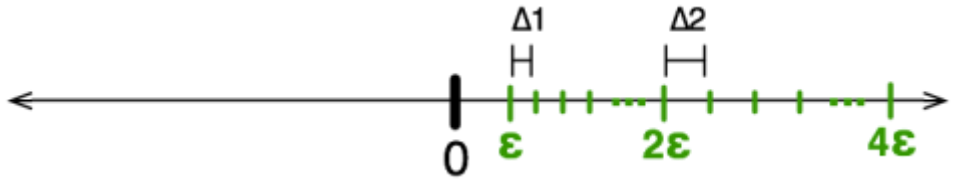
زمانی که Fraction به بیشترین مقدار خود برسد، ناچار به افزایش مقدار Exponent هستیم. پس بعد از این خواهیم داشت.

$$N = 1.0 * 2^{-2^{e-1}+1}$$

با افزایش Fraction در این فاز، تفاوت اعداد با یکدیگر برابر است با $2\Delta_1 = \Delta_2 = 2^{-f} * 2^{-2^{e-1}+1}$ و این به معناست که با افزایش Exponent، فاصله‌ی بین اعداد نیز افزایش پیدا می‌کند. به تعبیر دیگر هرچه از صفر فاصله بگیریم، دقت ما (فاصله‌ی بین اعداد) هم کاهش پیدا می‌کند. توجه کنید که این وضعیت برای اعداد منفی نیز به صورت قرینه خواهد بود، چون منفی بودن صرفا توسط بیت S مشخص می‌شود. به هر کدام از سطوح Exponent اصطلاحا یک اکتاو گفته می‌شود و فاصله‌ی بین اعداد در اکتاو i برابر خواهد بود با:

$$\Delta_1 = 2^{-f} * 2^{-2^{e-1}}, \Delta_i = 2^{i-1} * \Delta_1$$

اگر بخواهیم اعداد قابل نمایش توسط سیستم ممیز شناور را روی محور اعداد نمایش دهیم، خواهیم داشت:



مقایسه اولیه ممیز شناور و ممیز ثابت

- هزینه سخت‌افزاری مدارهای محاسباتی سیستم ممیز ثابت بسیار کمتر از سیستم ممیز شناور است.
- تاخیر محاسبات ممیز ثابت بسیار کمتر از تاخیر محاسبات ممیزشناور می‌باشد.
- نمایش اعداد در ممیزثابت در مقایسه با ممیز شناور، دارای انعطاف پذیری دلخواه نیست.
- سیستم ممیز شناور قابلیت نمایش اعداد خاص مثل e و π و $\sqrt{2}$ و.. .. را دارد.

محاسبات اعداد اعشاری ممیز شناور

برای انجام محاسبات جمع و تفریق اعداد اعشاری ممیز شناور ابتدا باید اعداد را هم‌نما کنیم.

$$2.344 \times 10^{-6}, 3.1415 \times 10^{2}$$

$$2.3446 \times 10^{-6}\ 314150000.0000 \times 10^{-6}$$

$$0.0000023446 \times 10^{2}\ 3.1415 \times 10^{2}$$

در هر دو حالت به علت محدودیت فضا ممکن است قسمتی از اعداد دور ریخته شود؛ در روش اول قسمت پر ارزش اعداد دور ریخته می‌شود و در روش دوم قسمت کم ارزش اعداد.

مثلا اگر کامپیوتری یک رقم قسمت صحیح و چهار رقم قسمت اعشاری عدد را ذخیره کند؛ فقط قسمت‌های رنگی اعداد بالا ذخیره می‌شود.

با این‌که هر دو روش از نظر ریاضی درست است اما به دلیلی که بالا گفته شد از روش دوم در کامپیوتر برای انجام محاسبات استفاده می‌شود.

الگوریتم جمع/تفریق اعداد اعشاری

۱. چک کردن صفر

ردیف کردن نما:

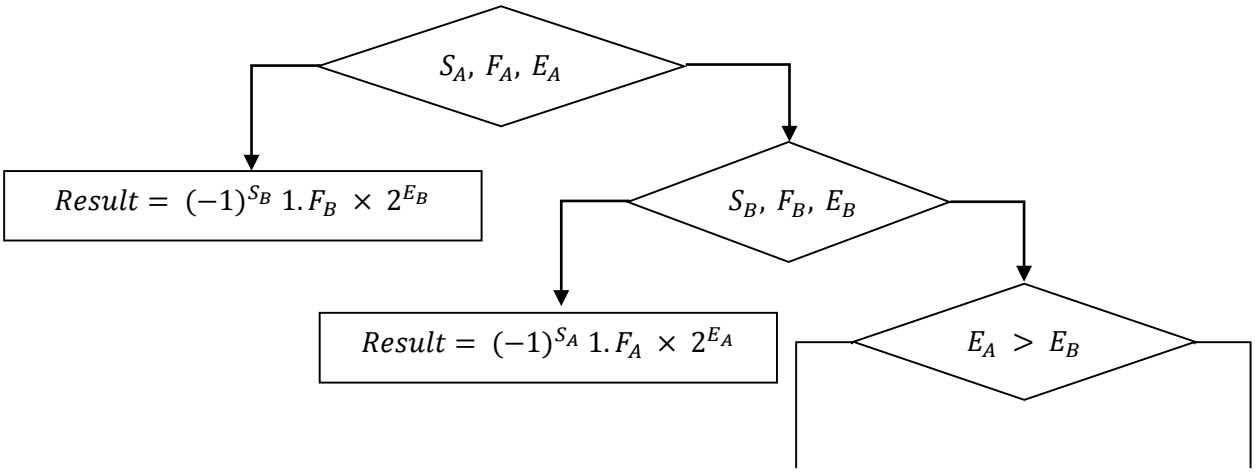
۲. مقایسه نماها (جهت پیدا کردن عدد با نمای بزرگتر)

۳. عدد با نمای کوچکتر را به اندازه نماها به سمت راست شیفت می‌دهیم.

۴. دو عدد را با هم جمع/تفریق می‌کنیم (جمع/تفریق کننده اندازه علامت)

۵. در صورتی که حاصل هنجار نباشد؛ آن را هنجار می‌کنیم.

$$A = (-1)^{S_A}\ 1.F_A \times 2^{E_A}, B = (-1)^{S_B}\ 1.F_B \times 2^{E_B}$$



در نمایش هنجار شده اعداد اعشاری یک رقم ۱ قبل از ممیز و بقیه ارقام بعد از ممیز قرار می‌گیرند.

$$(-1)^{S} 1.F \times 2^{E}$$

- زیر ریز (underflow): قبل از ممیز فقط رقم صفر وجود داشته باشد.
- سر ریز (overflow): قبل از ممیز عددی بزرگتر از ۱ وجود داشته باشد.

$$\begin{array}{r} 1.0011 \times 2^{10} \\ +\,1.0010 \times 2^{10} \\ \hline 10.0001 \times 2^{10} \end{array}$$

مثال۱: دو عدد $(0.5)_{10}$ و $(-0.4375)_{10}$ را با هم جمع کنید.

$$0.5 \times 2 = 1 \quad (0.5)_{10} = (1.000)_2 \times 2^{-1}$$

⟹

و به همین ترتیب داریم:

$$(-0.4375)_{10} = (-1.110)_2 \times 2^{-2}$$

حال الگوریتم جمع را اجرا می‌کنیم:

۱. *چک کردن صفر (که در اینجا نداریم!)*

۲. *مقایسه نماها(ارقام عدد با نمای کوچکتر را به اندازه نماها به سمت راست شیفت می‌دهیم):*

$(-1.110)_2 \times 2^{-2} = (-0.111)_2 \times 2^{-1}$

۳. *دو عدد را با هم جمع می‌کنیم:*

$(-0.111)_2 \times 2^{-1} + (1.000)_2 \times 2^{-1} = (0.001)_2 \times 2^{-1}$

۴. *در صورتی که حاصل هنجار نباشد؛ آن را هنجار می‌کنیم.*

الگوریتم ضرب اعداد اعشاری

۱. چک کردن صفر

۲. جمع نماها با هم $E_R = E_A + E_B - b$

۳. تعیین علامت حاصل $S_R = S_A \oplus S_B$

۴. انجام عمل ضرب $X = 1.F_A \times 1.F_B$

۵. هنجار کردن نتیجه در صورت ناهنجار شدن (overflow)

$(-1)^{S_R} X \times 2^{E_R}$

مثال۲: دو عدد $(0.5)_{10}$ و $(-0.4375)_{10}$ را در هم ضرب کنید.

مطابق مثال۱ داریم:

$(0.5)_{10} = (1.000)_2 \times 2^{-1}$

$(-0.4375)_{10} = (-1.110)_2 \times 2^{-2}$

حال الگوریتم ضرب را اجرا می‌کنیم:

۱. چک کردن صفر

۲. جمع نماها با هم: -1 + (-2) = -3

۳. تعیین علامت حاصل: $-1 \oplus 1 = -1$

۴. انجام عمل ضرب:

$$
\begin{array}{l}
\quad 1.000 \\
\times \quad 1.110 \\
\hline
0000 \\
1000 \\
1000 \\
\quad 1000 \\
\hline
1110000
\end{array}
$$

۵. هنجار کردن نتیجه در صورت ناهنجار شدن

حاصل $(1.110000)_2 \times 2^{-3}$ *است و از آنجا که ما به ۴ بیت برای نگهداری نیاز داریم:*

$$(1.1100)_2 \times 2^{-3}$$

الگوریتم تقسیم اعداد اعشاری

۱. چک کردن صفر

۲. تفریق نماها با هم $E_R = E_A - E_B + b$

۳. تعیین علامت حاصل $S_R = S_A \oplus S_B$

۴. انجام عمل تقسیم $X = 1.F_A \div 1.F_B$

۵. هنجار کردن نتیجه در صورت ناهنجار شدن (underflow)

$$(-1)^{S_R} X \times 2^{E_R}$$

در بعضی از سیستم‌ها برای نشان دادن ناعددی‌ها نمای اعداد اعشاری را با مقدار بایاس جمع می‌کنند.

در استاندارد IEEE754 برای دقت ساده از بایاس ۱۲۷ استفاده می‌شود. به عنوان مثال عدد ۱- به صورت زیر نمایش داده می‌شود:

$$-1 + 127 = 126 = (01111110)_2$$

همچنین عدد ۱ به صورت زیر نمایش داده می‌شود:

$$1 + 127 = 128 = (10000000)_2$$

توان بایاس در عدد ممیز شناور به صورت زیر نمایش داده می‌شود:

$(-1)^s \times 1.F \times 2^{Exponent-Bias}$

توان بایاس برای دقت مضاعف در این استاندارد ۱۰۲۳ است.

در مرحله دوم ضرب اعداد اعشاری، مقدار بایاس را از حاصل جمع نماها کم می‌کنیم تا مقدار بایاس را دوبار با حاصل جمع نکرده باشیم و در مرحله دوم تقسیم اعداد اعشاری، مقدار بایاس را با حاصل تفریق نماها جمع می‌کنیم تا مقدار بایاس را دوبار از حاصل کم نکرده باشیم. .

$$E_A = x_A + b, E_B = x_B + b \Rightarrow E_R = x_A + x_B + b = E_A + E_B - b$$
$$\Rightarrow E_R = x_A - x_B + b = E_A + E_B + b$$

مثال۳: عدد ۰. ۷۵- در مبنای ۱۰ را در نمایش باینری به صورت دقت ساده و دقت مضاعف نشان دهید.

برای این که عدد ۰. ۷۵- را به مبنای دو ببریم باید آن را در ۲ ضرب کنیم. قسمت صحیح عدد به دست آمده را نگه می‌داریم و قسمت اعشاری را دوباره در ۲ ضرب می‌کنیم و این عمل را تا زمانی ادامه می‌دهیم که عدد بدست آمده قسمت اعشاری نداشته باشد:

$0.75 \times 2 = 1.5$ $0.1\ldots$

$\Longrightarrow$

$0.5 \times 2 = 1$ 0.11

$\Longrightarrow$

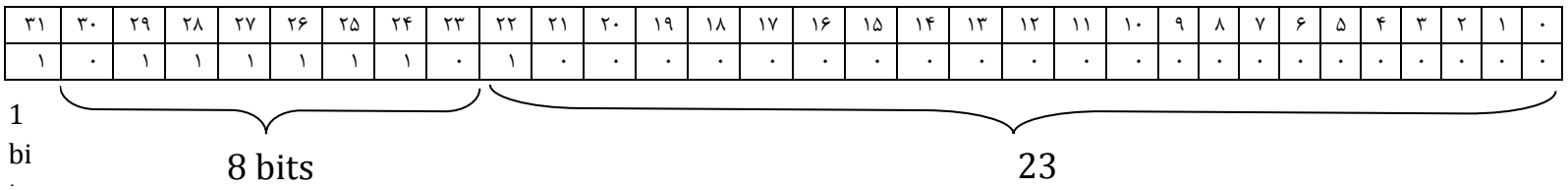
و چون عدد ما منفی است جواب ۰. ۱۱- در مبنای ۲ می‌باشد. حال عدد بدست آمده را هنجار می‌کنیم: -1.1×2^{-1}

نمایش عمومی دقت ساده به صورت زیر است:

$$(-1)^{s} \times 1.F \times 2^{Exponent-127}$$

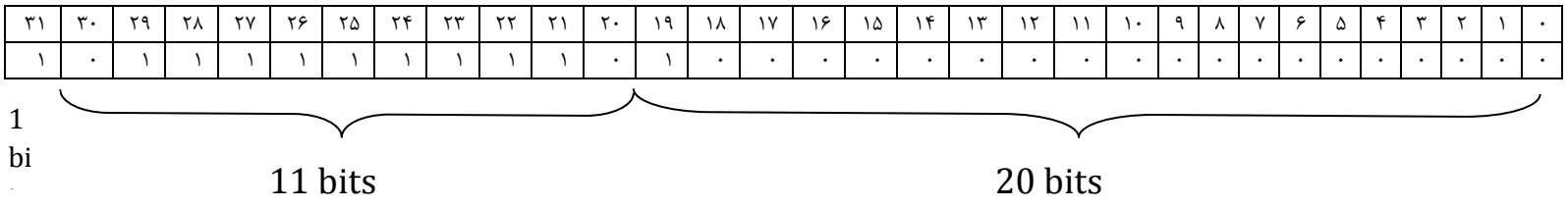
که به صورت زیر جایگزین می‌شود:

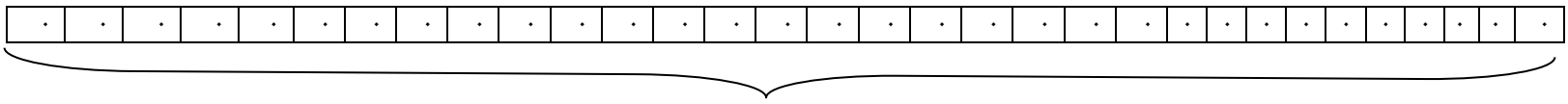
$$(-1)^{1} \times 1.1000\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{126-127}$$



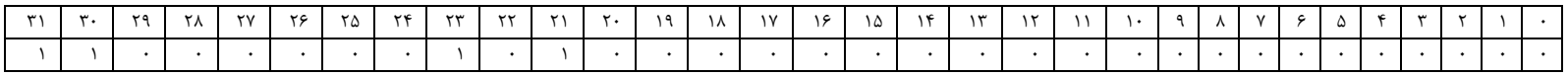
همچنین نمایش دقت مضاعف به صورت زیر است:

$$(-1)^{1} \times 1.1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{1022-1023}$$





مثال۴: عدد ممیز شناور زیر در دقت ساد سیمالی است؟



بیت علامت ۱ است. محدوده‌ی توان شامل ۱۲۹ است و محدوده‌ی اعشار در مبنای دودویی شامل ۰. ۰۱ می‌باشد:

$$1 \times 2^{-2} = \frac{1}{4} = 0.25$$

بنابراین عدد موردنظر به صورت زیر نمایش داده می‌شود:

$$(-1)^{1} \times 1.25 \times 2^{129-127} = (-1)^{1} \times 1.25 \times 2^{2} = \textit{-5.0}$$

نمایش BCD (دهدهی کدشده به صورت باینری Binary Coded Decimal)

در این سیستم هر رقم با ۴ بیت نمایش داده می‌شود و بخاطر ارتباط با دستگاه‌های I/O هنوز از آن استفاده می‌شود.

محاسبات بر مبنای نمایش BCD

جمع BCD

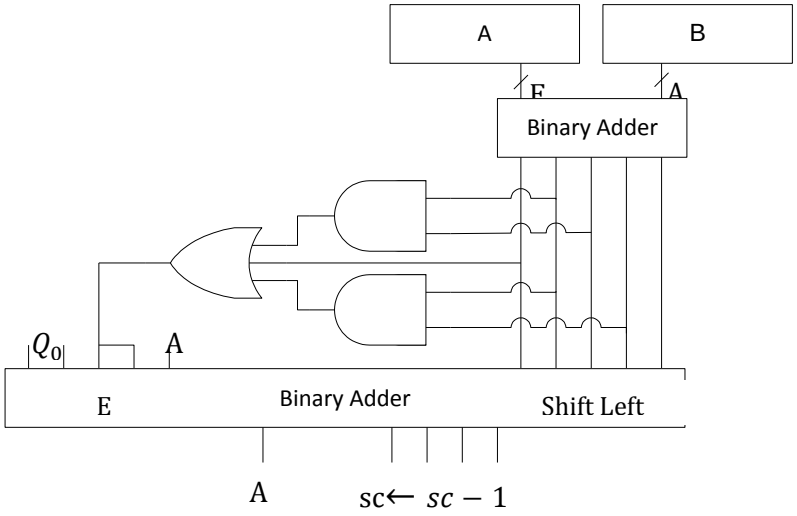
می‌خواهیم جمع کننده‌ای بسازیم که دو رقم BCD را با هم جمع کند.

باید مداری طراحی کنیم که اگر حاصل دو رقمی شد آن را با ۶ جمع کند. پس باید مداری برای تشخیص آن‌که آیا عدد دورقمی است یا نه نیز داشته باشیم (F خروجی این تابع است)

$$\begin{array}{r} a_3\ a_2\ a_1\ a_0 \\ +\ b_3\ b_2\ b_1\ b_0 \\ \hline c_B\ s_3\ s_2\ s_1\ s_0 \end{array} \Longrightarrow c_D\ s_3'\ s_2'\ s_1'\ s_0'$$

$$F = c_B + s_3 s_2 + s_3 s_1$$

	c_B	$s_3 s_2 s_1 s_0$	c_D	$s_3' s_2' s_1' s_0'$	F
0	0	0000 0	0	0000 0	0
1	0	0001 1	0	0001 1	0
2	0	0010 2	0	0010 2	0
3	0	0011 3	0	0011 3	0
4	0	0100 4	0	0100 4	0
5	0	0101 5	0	0101 5	0
6	0	0110 6	0	0110 6	0
7	0	0111 7	0	0111 7	0
8	0	1000 8	0	1000 8	0
9	0	1001 9	0	1001 9	0
10	0	1010 10	1	0000 0	1
11	0	1011 11	1	0001 1	1
12	0	1100 12	1	0010 2	1
13	0	1101 13	1	0011 3	1
14	0	1110 14	1	0100 4	1
15	0	1111 15	1	0101 5	1
16	1	0000 0	1	0110 6	1
17	1	0001 1	1	0111 7	1
18	1	0010 2	1	1000 8	1



مـدار بـالا یـک BCD H. A و بـا cascade کردن آن‌ها می‌تـوان یـک جمع کننده ساخت.

تفریق BCD

می‌توان بجای محاسبه A-B مقدار A+10's(B) را حساب کرد.

A	10's(A)
0	10
1	9
2	8
3	7
4	6
5	5
6	4
7	3
8	2
9	1
10	0

اگر حاصل carry ایجاد کند؛ حاصل مثبت است و carry را دور می‌ریزیم و اگر حاصل carry ایجاد نکند؛ حاصل عددی منفی و به صورت مکمل ۱۰ است.

$$A-B \Rightarrow \begin{cases} c=0 & A<B \\ c=1 & A\geq B \end{cases}$$

مدار محاسبه‌کننده مکمل۱۰، ۴بیت ورودی می‌گیرد و ۵ بیت خروجی می‌دهد.

ضرب BCD

ابتدا حالت ساده‌ای از ضرب BCD را بررسی می‌کنیم که در آن دو عدد تک رقمی را در یکدیگر ضرب می‌کنـیم برای مثال:

1001 × 0100 =0011 0110

```
      1001 = 9
×     0100 = 4
-------------------
0010 0100 = 24 (اشتباه)
```

```
0010 0100 = 24 (اشتباه)
÷ 1010 = 10
-------------------
0011 = 3 خارج قسمت
0110 = 6باقیمانده
```

حال که ضرب یک رقم در یک رقم را محاسبه نمودیم ضرب یک رقم در چند رقم را مورد بررسی قرار می‌دهیم:

0011 1001 = 39	1001	0011
× 0100 = 4	× 0100	× 0100
?	0011 0110	0001 0010

ابتدا حاصل هر رقم را با ارقام دیگر به دست می‌آوریم و سپس حاصل را با یکدیگر جمع می‌کنیم

```
 0011 1001 = 39
× 0100 = 4
-------------------
      0011 0110
+ 0001 0010
-------------------
0001 0101 0110 =156
```

حال ضرب چند رقم در چند رقم را همانند ضرب در دنیای واقعی محاسبه می‌کنیم

```
 0011 1001 = 39
× 0100 0100 = 84
-------------------
      0001 0101 0110 =156
+ 0011 0001 0010     =312
-------------------
 0011 0010 0111 0110=3276
```

تقسیم BCD

می توان با تبدیل عدد به باینری عمل تقسیم را انجام داد و سپس با تقسیم بر ۱۰ رقم‌های آن را جدا نمود

مثال:

0101 0010 ÷0100 = 0001 0011
(52 ÷ 4 = 13)

```
     0101     = 5
×    1010     = 10
-------------------
0011 0010     = 50(10)
```

```
0011 0010     = 50(10)
+    0010     = 2
-------------------
0110 0100     = 52(10)
```

```
0110 0100     = 52(10)
÷    0100     = 4
-------------------
①    1101     = ?      (اشتباه)
```

```
0000 1101     = ?      (اشتباه)
÷    1010     = 10(10)
-------------------
     0001     = 1
   با باقیمانده
```

$\Rightarrow$ 1 3
0001 0011

0011 = 3

باقیمانده تقسیم را هم از مرحله① به دست می‌آید که با باقیمانده آن نیز به همان شکل برخورد می‌شود و با تقسیم بر ۱۰ رقم‌های آن را جدا می‌کنیم

تمرین:تقسیم BCD را بدون تبدیل به عدد باینری متناظر محاسبه کنید. (راهنمایی: همانند تقسیم در دنیای واقعی عمل تقسیم را انجام دهید)

فصل سوم

Control Unit (واحد کنترل)

در این بخش شما یاد می‌گیرید چگونه واحد کنترل پردازشگر را طراحی نمایید.

در ابتدا شما با وظیفه واحد کنترل آشنا می‌شوید که برای کار خود مجبور است از یک الگوریتم ترتیبی(فن نیومن) استفاده کند این تنها جایی است که در سخت افزار ترتیبی عمل می‌شود و این باعث کاهش زیاد سرعت سخت افزار می‌شود که برای رفع این مشکل از تکنیک خط لوله استفاده می‌شود.

سپس شما با انواع واحدهای کنترل آشنا می‌شوید که شامل واحد کنترل برنامه پذیر و واحد کنترل سیم بندی شده است. که هرچند واحد کنترل سیم بندی شده قابلیت تغییر ندارد و نظم کمتری دارد اما به علت سرعت بسیار بسیار بالای آن نسبت به واحد کنترل برنامه پذیر در ساخت پردازشگرها از آن استفاده می‌شود.

در انتهای فصل با مبحث ورودی و خروجی‌ها آشنا می‌شویم و نحوه ارتباط پردازشگر با IO مورد بررسی قرار می‌گیرد و سپس برای سرعت بخشیدن به کار پردازشگر از وقفه‌ها (Interupt) استفاده می‌شود.

در پایان این بخش شما قادر خواهید بود هر نوع پردازشگری بسازید.

CPU بـه عنـوان ورودی مـی توانـد مجموعـه ای از دسـتورالعمل‌ها را بگیـرد کـه بـه آن Instruction Set Architecture (ISA) می گویند. برای طراحی یک پردازنده با مجموعه دستورات خاص باید به چند نکته توجه کنیم:

۱. کامپیوتر RISC است یا CISC؟

برای پاسخگویی به این سوال باید دید که ماهیت دستورالعمل‌ها چیست:

a. I/O

b. MEM

i. ALU Based

ii. Memory Based

۲. تعداد دستورالعمل‌ها چیست؟

۳. تنوع دستورها چگونه است؟

مثلا دستوراتی که با حافظه کار می کنند چگونه اند و Operand دستورالعمل‌ها کجاسـت. آیـا در داخـل خـود دستورات به صورت ضمنی است؟ آیا در ادامه دستورالعمل آمده اند؟ و یا مثلا در Stack هستند؟

۴. تعداد Operand دستورالعمل‌ها چندتاست؟ (پردازنده چند آدرسه است؟)

۵. شیوه های آدرس دهی چگونه اند؟

انواع ماشین ها:

- صفر آدرسی (پشته ای): دستورالعمل‌ها فاقد Operand مشخص هستند و ماشـین بـا Stack کـار مـی کند. مثلا:

۱۰
۴
۳
۲

(۱۰+۴-۳)/۲

Add , Sub , Div

- تک آدرسی: دستورالعمل‌ها یک Operand می گیرند.
- دو آدرسی: دستورالعمل‌ها دو Operand می گیرند.
- سه آدرسی: دستورالعمل‌ها سه Operand می گیرند.
- . . .

قاعدتا هرچه تعداد Operandها بیشتر شود خصوصیات کامپیوتر به CISC نزدیکتر می شود.

تعداد دستورالعمل‌ها نیز مهم است و اثر مستقیم بر مبنای دستورات دارد. به علاوه در طراحی CPU باید قالب دستورات (Instruction Format) را مشخص کرده باشیم و تمام دستورات را بر مبنای آن بیان کنیم.

قالب دستورات:

Operand 3	Operand 2	Operand 1	Opcode	شیوه آدرس دهی

اگر ماشین صفر آدرسه باشد. دستورالعمل‌ها فقط از Opcode تشکیل می شوند.

در حالت کلی دستورات اجزای ثابتی ندارند مثلا در تعداد Operandها متفاوتند. اما با این اوصاف قالب آنها باید یکتا باشند تا با هم تداخل نکنند.

مشخص است که پهنای Opcode حداقل $\lceil Log\ k \rceil$ است که k تعداد دستورات است.

می توان برای مشخص کردن صفر آدرسه یا تک آدرسه بودن دستور از یک بیت flag استفاده کرد اما بهتر است که از ترکیب بیت‌ها استفاده شود تا حجم دستورات کاهش یابد.

شیوه های آدرس دهی:

۱. ضمنی (Implicit):

مثلا: STD → Set Direction Flag

CLC → Clear Carry

اولین کامپیوترها با این نوع آدس دهی کار می کردند اما برنامه نویسی در آنها بسیار محدود بود.

۲. بلافاصله (Immediate)

مثلا: Add 5 Mul 10

این روش باز هم محدود کننده است و از آن فقط برای کارهای Static می توان استفاده کرد.

۳. حافظه ای مستقیم (Memory Direct):

مثلا: Add [6]

یعنی در حافظه عدد موجود در آدرس ۶ را جمع بزن. در این حالت، داده در دل دستورالعمل نهفته است.

در این نوع آدرس دهی نمی توان با Pointer کار کرد.

Flag	*ADD*	*6*

۴. حافظه ای غیر مستقیم (Memory Indirect)

مثلا: MUL [[s]]

که S آدرس آدرس داده مورد نظر ما است.

اشکال این روش وقت گیر بودن آن است چرا که نیاز به دو مرتبه دسترسی به حافظه در آن وجود دارد. به منظور رفع این مشکل از انواع آدرس دهی ثباتی استفاده می شود.

۵. ثباتی مستقیم

این نوع آدرس دهی برتری نسبت به حافظه ای مستقیم ندارد و همچنین دارای محدودیت تعداد ثبات‌ها نیز هست. مثلا:

ADD BX

۶. ثباتی غیرمستقیم

با استفاده از این نوع آدرس دهی عملیات های آرایه ای بسیار سریع تر می شوند.

می توان با استفاده از این نوع آدرس دهی پشته را پیاده سازی کرد.

MUL [BX]

۷. آدرس دهی نسبی با ثبات پایه (Base Addressing)

معمولا کامپیوترها برای این نوع آدرس دهی ثباتی با نام ثبات پایه دارند. مثلا:

ADD BX[4]

با استفاده از این روش می توان نقطه شروع داده ساختار را به جای دیگری منتقل کرد و از دوباره کاری جلوگیری کرد.

۸. شاخص دار (Index)

مثلا: ADD Value [SI]

این نوع آدرس دهی هنگامی استفاده می شود که Offset متغیر است. به همین منظور بیشتر برای پیمایش آرایه از آن استفاده می شود. از ترکیب این روش با روش قبلی برای کار با آرایه های دو بعدی استفاده می شود.

۹. آدرس دهی شاخص دار با ثبات پایه

مثلا:ADD BX[SI]

از این شیوه‌ی آدرس دهی برای کار با آرایه دو بعدی استفاده می‌شود.

۱۰. آدرس دهی خودافزایشی(خودکاهشی)(Aouto Increment,Aout Decrement)

مثلا: ADD +[R]

با توجه به ISA داده شده هدف ما طراحی پردازنده RISC ۱۶ بیتی است که در آن تنها از شیوه‌های آدرس دهی حافظه‌ای مستقیم، حافظه‌ای غیرمستقیم و ثباتی مستقیم استفاده می‌شود.

هم چنین با کمک Instruction Set داده شده باید IF و حداقل تعداد ثبات‌های مورد نیاز را معین کنیم. مثلا تعیین کنیم که برای انجام اعمال مورد نظر چه ثبات‌هایی نیاز داریم، آیا به Temp Register نیازمندیم یا خیر؟ و. ..

سپس باید کار طراحی را انجام دهیم. طراحی شامل دو بخش است:

۱. طراحی Data Path (مسیر داده)

۲. طراحی Control Unit (واحد کنترل)

در مسیر داده باید تمام اطلاعات مربوط به همه‌ی چیزهایی که با داده ارتباط دارند را طراحی کنیم. این که داده از کجا می‌آید، در چه مسیری به کجا می‌رود و. .. . یعیی فرآیند حرکت داده را باید مدیریت و کنترل کنیم.

به طور خلاصه در مسیر داده همه‌ی فرآیندهای مربوط به هر مولفه‌ای که برای نگهداری، انتقال، انتخاب، دسترسی و عبور داده به کار می‌رود، باید طراحی شود.

در واحد کنترل نیز هر مولفه‌ای که وظیفه‌ی کنترل، انتخاب عملیات، هدایت داده و اجرای دستورالعمل را بر عهده داشته باشد باید طراحی شود.

مسیر داده شامل: ALU , Register file , Registers , BUS می‌باشد.

واحد کنترل شامل: SC(Sequence counter) , Decoders , Multiplexers , flags و سخت افزار کنترل کننده می‌باشد.

فرآیند کار در پردازنده بدین ترتیب است که: ابتدا دستورالعمل از حافظه وارد CPU می‌شود، سپس Decode می‌شود، پس از آن Command به ALU داده می‌شود و در صورت وجود Operand، از حافظه خوانده می‌شود، سپس دستور اجرا می‌شود و خروجی‌ها در جای مناسب خود قرار می‌گیرند.

به این روند الگوریتم فون نیومن (Von Neumann) می‌گویند، که به شرح زیر می‌باشد:

۱. خواندن دستورالعمل از حافظه(واکشی دستورالعمل) Instruction Fetch

۲. بازگشایی دستورالعمل Instruction Decode

۳. خواندن اپرندهای دستورالعمل Operands Fetch

۴. اجرای دستورالعمل Execute

۵. بازنویسی نتیجه Write Back Result

۶. $PC \leftarrow PC + 1$

۷. برو به ۱

این جا اولین جایی است که ناچاریم در سخت افزار به صورت Sequential عمل کنیم. و در هر مرحله برخی از قسمت‌های سخت افزار بلا استفاده اند. از طرف دیگر هر چه عمق یک مرحله زیاد شود، اجرای برنامه کندتر می‌شود.

SC: یک شمارنده است که به ما می‌گوید در کدام یک از گام‌های الگوریتم هستیم.

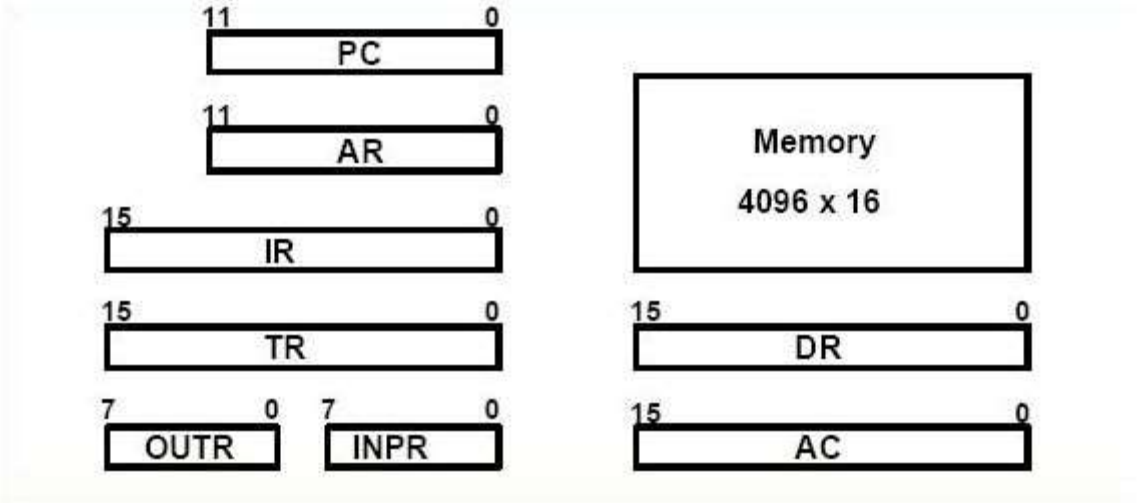
Instruction Cycle: به هر بار اجرای الگوریتم فون نیومن از ابتدا تا انتها Instruction Cycle می‌گویند.

همان طور که پیش از این هم گفتیم می‌خواهیم پردازنده RISC ۱۶ بیتی طراحی کنیم که در آن سه نوع دستورالعمل: حافظه‌ای، ثباتی و I/O وجود دارد.

دستورات حافظه‌ای درمورد داده‌های داخل حافظه است و دو شیوه‌ی آدرس دهی مستقیم و غیرمستقیم برای آن به کار می‌رود.

دستورات ثباتی نیز دارای شیوه‌ی آدرس دهی ثباتی هستند.

با توجه به Instruction Set درمورد حافظه و ثبات‌های مورد نیازمان تصمیم گیری می‌کنیم :

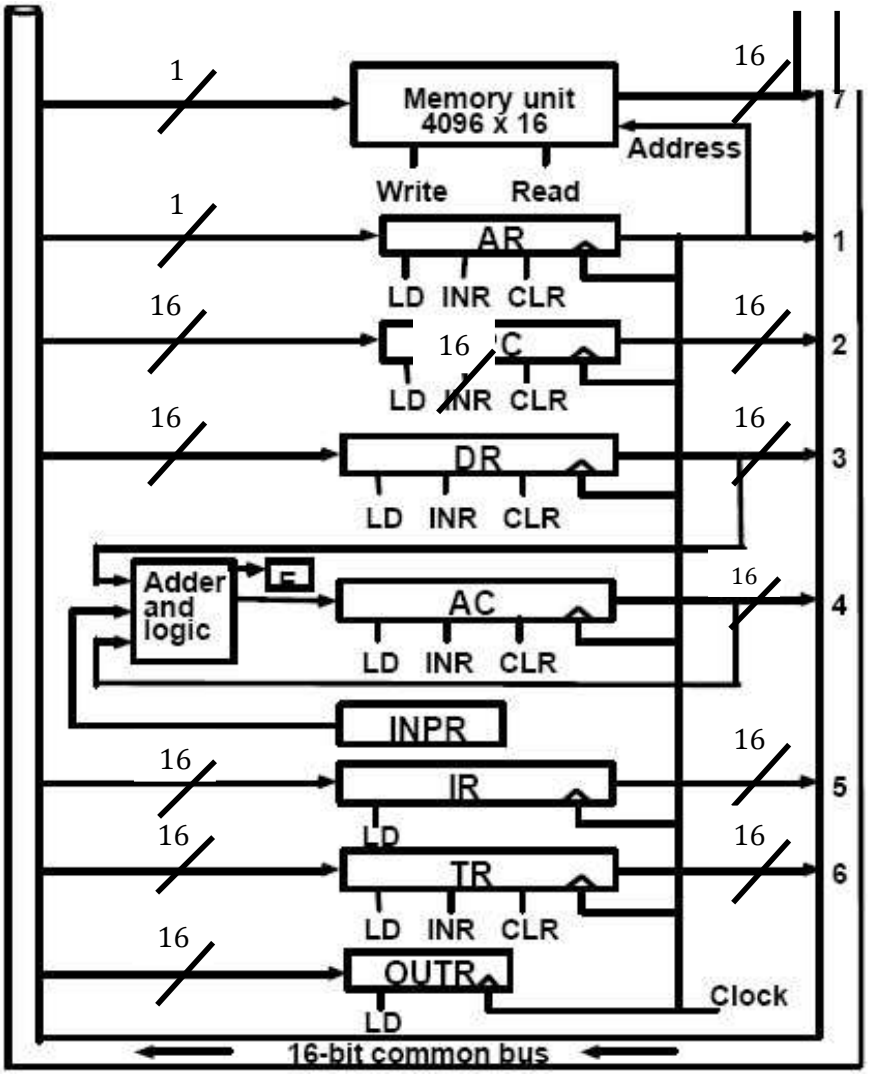


کاربرد این ثبات‌ها به شرح زیر است:

- PC(Program Counter)(۱۲ بیت): آدرس دستورالعمل در حافظه را نگه داری می‌کند.
- AR(Address Register) (۱۲ بیت): آدرسی از حافظه را نگه داری می‌کند.
- DR(Data Register) (۱۶ بیت): داده‌هایی که از Data Bus می‌آیند را نگه می‌دارد.

- IR(Instruction Register) (۱۶ بیت):دستورالعمل‌هایی که PC در حافظه به آن‌ها اشاره می‌کند در این رجیستر نگه داری می‌شود.
- AC(Accumulator) (۱۶ بیت): خروجی ALU در این ثبات نگه داری می‌شود. (بهتر است که خروجی ALU در ثبات نگه داری شود چرا که نتیجه‌ی محاسبه‌ی ALU ممکن است یک نتیجه میانی باشد. از این رو خروجی ALU به AC می‌رود و ورودی‌های آن نیز AC,DR می‌باشند.)
- TR(Temp register) (۱۶ بیت): نتایج میانی و. .. را نگه داری می‌کند.

برای ارتباط رجیسترها با یکدیگر از شیوه‌های متفاوتی استفاده می‌شود. مثلا: استفاده از Bus یا به صورت Point to Point. اما متداول ترین شیوه استفاده از Bus است. Bus به صورت زیر قرار داده می‌شود:



برای اعمال ورودی و خروجی، رجیسترهای INPR و OUTR را در نظر می‌گیریم. رجیسترهای ورودی و خروجی برای سادگی ۸بیتی در نظر گرفته شده‌اند. به عنوان مثالی از دستورات ورودی و خروجی در صورتی که دستور، OUT 50 باشد، یعنی داده‌ی درون OUTR را به پُرت شماره‌ی ۵۰ بفرست.

برای باس، ۱۶ بیت در نظر گرفته شده است که در صورتی که از ۱۲ بیت آن استفاده شود، مقداری که روی باس قرار دارد از جنس آدرس است و در صورتی که از ۱۶ بیت استفاده شود، از جنس داده است. هم‌چنین می‌توان به جای یک باس از دو باس مجزا برای آدرس و داده استفاده کرد.

در ALU که برای این کامپیوتر پایه استفاده می‌کنیم، اعمال ضرب و تقسیم در نظر گرفته نشده است و از عملیات حسابی تنها عمل جمع و تفریق قابل انجام است. عملیات منطقی هم همان‌طور که انتظار می‌رود در ALU انجام‌پذیر است. ورودی‌های ALU را DR، INPR و AC تشکیل می‌دهند.

در صورتی که تمام خروجی‌های رجیسترهای مختلف، مستقیما به باس متصل شوند، به علت اتصال خروجی‌ها به هم، باید از واسطی استفاده کرد. به این منظور سه راه وجود دارد:

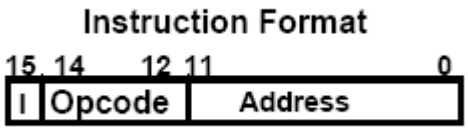
۱. MUX
۲. Tri-State
۳. Bus Driver: در صورتی که سیگنال این واسط صفر باشد، اتصال ورودی و خروجی قطع است.

در کامپیوتر پایه برای حل این مشکل از مالتی‌پلکسر استفاده می‌شود. به این صورت که تمام خروجی‌های رجیسترها و حافظه به عنوان ورودی به MUX می‌روند. برای این مالتی‌پلکسر سه ورودی انتخاب در نظر گرفته می‌شود.

هرگاه می‌خواهیم از اطلاعات حافظه استفاده کنیم، باید در AR، آدرس را بنویسیم.

انواع دستورات:

دستورات حافظه‌ای تک‌اپرندی هستند و دستورات ثباتی یک یا دو اپرندی هستند. با توجه به شیوه‌های آدرس‌دهی حافظه‌ای مستقیم و غیرمستقیم، اپرند یک آدرس بوده و ۱۲ بیتی است.



به این ترتیب، OpCode ۸ حالت دارد. برای دستورات حافظه‌ای از قبیل load، store، jump و. .. از ۷ تای اول آن استفاده می‌کنیم. یعنی:
000, 001, 010, 011, 100, 101, 110

دستورهایی که ۴ بیت سمت چپ آنها 0111 است را برای دستورهای ثباتی و دستورهایی را که ۴ بیت سمت چپ آنها 1111 است را برای دستورهای I/O استفاده می‌کنیم.

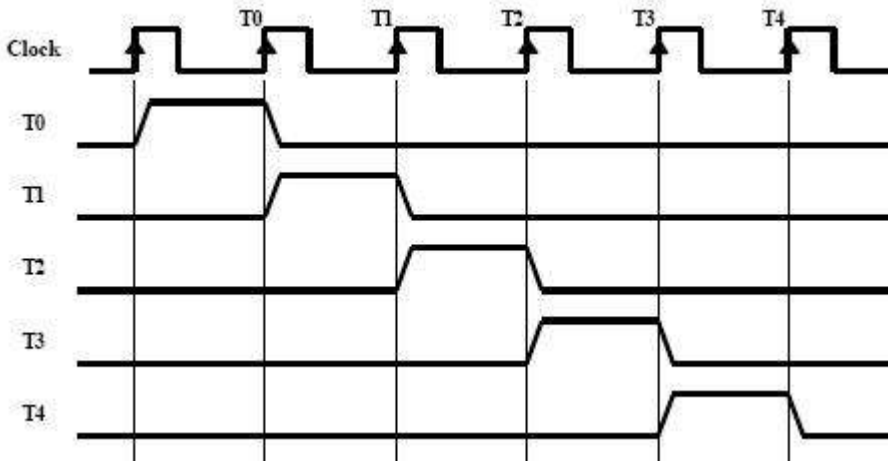
از آنجا که تنها ۱۲ دستور ثباتی در کامپیوتری که می‌خواهیم طراحی کنیم وجود دارد، کافی است در ۱۲ بیت باقی‌مانده برای هر دستور یک بیت را ۱ و باقی بیت‌ها را صفر قرار دهیم.

در دستورات I/O، یک بیت را برای تمایز بین دستورهای ورودی و خروجی در نظر می‌گیریم و باقی را برای شماره‌ی پُرت می‌گذاریم.

طراحی واحد کنترل:

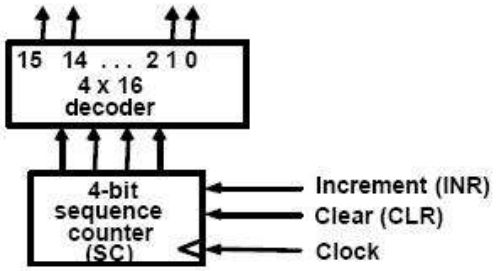
وظیفه‌ی طراحی پایه‌های load و تنظیم ورودی‌های انتخاب برای مالتی‌پلکسر باس، در واحد کنترل انجام می‌شود.

برای رعایت الگوریتم فون نیومن و کار کردن ترتیبی با سخت‌افزار، باید سخت‌افزاری ارائه کنیم که توالی بدهد. احتیاج به شمارنده‌ای داریم که به این شکل باشد:



برای تولید این شکل موج می‌توان از shift-register استفاده کرد. به این صورت که یک بیت ثبات را برابر ۱ کرده و باقی بیت‌ها را صفر می‌گذاریم. و با هر بار شیفت دادن این ثبات، عدد یک به بیت‌های مختلف ثبات می‌رسد و در هر لحظه تنها یک بیت ثبات ۱ است. به این روش بیت لغزان می‌گویند.

یک شیوه‌ی دیگر برای تولید شکل موج‌های گفته شده، استفاده از یک شمارنده‌ی صعودی و یک decoder متصل به این شمارنده است. در پایان هر مرتبه انجام الگوریتم فون‌نیومن، شمارنده را صفر می‌کنیم.



یکی از شیوه‌های توصیف الگوریتم فون‌نیومن، استفاده از micro-instructionها است. یک micro-instruction عبارتی به شکل زیر است:

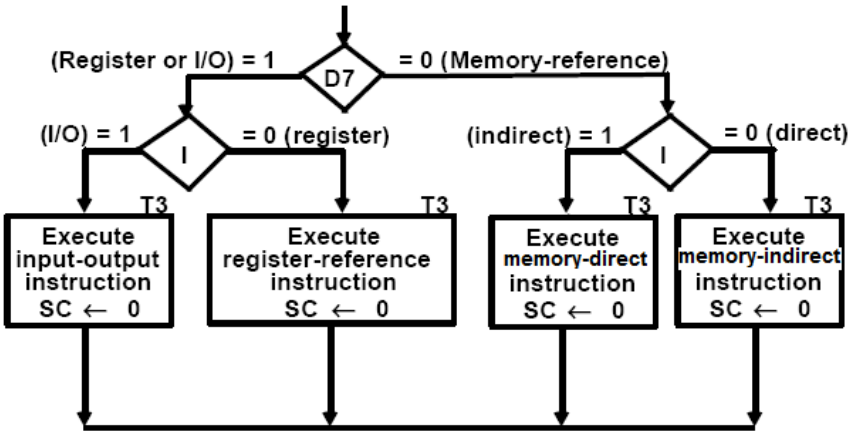
k: r ← s + t

به این معنی که در صورت برقراری شرط k، حاصل جمع s و t را در رجیستر r بریز.

عمل fetch در T0 و T1 انجام می‌شود:

T0: AR ← PC
T1: IR ← M [AR], PC ← PC + 1
decode:
T2: D0,. .. , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

و در ادامه داریم:



هر کدام از جعبه‌های بالا در T3 شروع به کار می‌کنند. برخی با یک کلاک انجام می‌شوند اما بعضی مثل حافظه-ای مستقیم یا غیر مستقیم بیش از یک کلاک طول می‌کشند.

به این سیکل که از fetch شروع می‌شود و به اجرای دستور ختم می‌شود، instruction cycle می‌گویند. (machine cycle با این سیکل تفاوت داشته و همان clock پردازنده است.)

فرض می‌کنیم input فقط صفحه کلید و output فقط چاپگر است و دستورات IN و OUT شماره پورت می-گیرند. برای ارتباط با دستگاه‌های وروی و خروجی از دو رجیستر flag به نام‌های FGI و FGO استفاده می‌کنیم تا با استفاده از آن، از مکانیزم polling استفاده می‌شود.

برای مالتی پلکسر انتخاب کننده‌ی ثبات‌ها باید شماره‌هایی به ثبات‌ها داده شود. از ۱ تا ۷ به آن‌ها شماره می‌دهیم.

دستورات حافظه‌ای:

۱. BUN: Branch Unconditionally
به صورت غیر مستقیم، دستور jump اجرا می‌شود.

۲. BSA: Branch and Save Return Address
با اجرای این دستور، علاوه بر اجرای دستور jump، آدرس دستور بعدی نیز ذخیره و نگه داری می‌گردد. در این نوع کامپیوتر پشته(stack) نداریم. پس می‌توان آدرس را در یک ثبات گذاشت. اما در این صورت فقط یک بار می‌توان این دستور را صدا کرد.

*قرارداد: وقتی یک تابع می‌نویسیم، خط اول تابع را خالی می‌گذاریم و آدس بازگشت را در همان خط خالی اول تابع ذخیره می‌کنیم.

۳. IBUN
برای بازگشت(return) از تابع استفاده می‌شود.

۴. ISZ: Increment and Skip if Zero
ابتدا AC یک واحد افزوده می‌شود، سپس اگر صفر شد، دستور بعد اجرا نمی‌شود.

با استفاده از این دستورالعمل‌ها، می‌توان همه‌ی برنامه‌ها غیر از برنامه‌های بازگشتی را نوشت.

زمانی که دستور HLT اجرا شود، CPU با این که منبع تغذیه ندارد اما هیچ کاری نمی‌کند مگر این که به صورت سخت افزاری reset شود. در مواقعی که سیستم عامل نداریم و اجرای برنامه تمام شده باشد، نکته‌ی مفیدی محسوب می‌گردد.

برای وقفه‌ها هم، یک flag به نام IEN در نظر می‌گیریم و فرض می‌کنیم CPU نیز یک پایه‌ی سخت افزاری برای interrupt دارد. وقتی این وقفه اعمال شود، یک فلیپ فلاپ به نام R، ۱ می‌شود. سپس subroutine مربوط به وقفه اجرا می‌شود که در قسمت اول حافظه interrupt vector می‌آید و مشخص می‌کند که روتین وقفه از کجا آغاز می‌گردد.

در این کامپیوتر فقط یک وقفه داریم که آدرس آن در خانه‌ی ۱ از حافظه نوشته می‌شود.

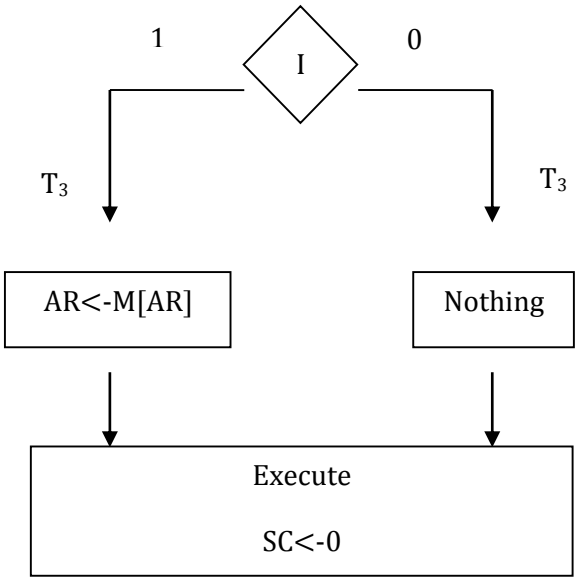
به طور کلی دو نوع طراحی داریم:

- ✓ طراحی Hardwired یا سیم بندی
- ✓ طراحی microprogrammed یا ریزبرنامه ریزی شده

در طراحی hardwired با گیت‌ها سر و کار داریم. اما با یک بار طراحی IC همیشه یک جور کار می‌کند و اگر بعدها دستوری اضافه شود، باید از اول طراحی را انجام داد. در صورتی که در طراحی microprogrammed برنامه ریزی می‌کنیم و عملا یک CPU در داخل CPU دیگراست. بنابراین با تغییر دستورالعمل‌ها کافیست برنامه ریزی را تغییر دهیم.

در CU ورودی‌های T_0 تا T_{15} و به علاوه opcode و I وارد می‌شود.

اما برای آدرس دهی فاصله‌ای مستقیم و غیر مستقیم به صورت فلوچارت زیر عمل می‌کنیم:



اگر به جای Nothing در حالت مستقیم در T_3 کار اجرا را شروع می‌کردیم، سخت افزار پیچیده تر می‌شد و در عمل بایستی سخت افزار مشابه داشته باشیم که یکی در T_3 و دیگری در T_4 فعال شود.

$\overline{D}_7 I T_3 : AR \leftarrow M[AR]$

$\overline{D}_7 \bar{I} T_3 : Nothing$

$D_7 \bar{I} T_3 : Execute\ a\ Register\ Reference\ Inst.$

$D_7 I T_3 : Execute\ an\ Input - Output\ Inst.$

در این حالت و با این شرط‌ها کارهای مختلف از هم جدا می‌شوند. I در دستورات بالا باید باشد. چرا که دو تا از اعمال هجرا می‌شود و هر دو با MUX در خروجی می‌آیند و لذا IC می‌سوزد.

باید دقت شود که وقتی opcode دستورات حافظه‌ای decode شده‌اند، اجرای دستورات در زمان‌های معین انجام شود و در دستور ذکر شود، مثلا: $D_0 T_4$

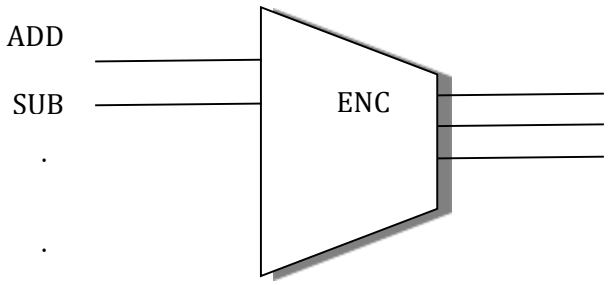
هم چنین در هنگام کار با ALU هر دو اپرند در AC و DR باشند. بعد از کار ALU با D_0، کار تمام است پس در این هنگام SC باید ۰ گردد.

برای هر ثبات مثلا AC باید پایه‌های LD و INC و CLR را زمانبندی کنیم. برای هر پایه تمام دستوراتی که مثلا AC مقصد است را نگاه می‌کنیم. برای مثال تمام حالاتی که باعث clear شدن می‌شوند را OR کرده و به پایه‌ی

CLR می‌دهیم. بدین ترتیب پایه‌های LD و INC و CLR برنامه ریزی شده و سخت افزار آن به دست می‌آید. برای خواندن/نوشتن از/در حافظه، نیز به همین صورت عمل می‌کنیم.

برای $S_0S_1S_2$ باید ثبات‌هایی که سمت راست ریزعمل است را در نظر بگیریم. قبل از $S_0S_1S_2$ هم یک Encoder قرار می‌دهیم تا هر سه مقدار بگیرند.

برای commandهای ALU هم به طور مشابه عملیات سمت راست micro-operation را نگاه می‌کنیم و حالاتی که به ALU نیاز داریم را تولید می‌کنیم.



خط لوله[۲۶]

پردازش موازی:

پرادزش موازی به معنی بکارگیری تکنیکهای متنوعی در پردازش همزمان داده است ک به منظور افزایش سرعت محاسبات سیستم‌های کامپیوتری مورد استفاده قرار گرفته اند.

پایین ترین سطح بررسی پردازش موازی، بررسی آن از نظر ثبات‌های بکار گرفته شده است.

دیدگاه‌های مختلف تقسیم بندی پردازش موازی:

- سازمان داخلی
- اتصالات درونی بین پرازنده ها
- چریان اطلاعات درون سیستم

---

۲۶ PipeLine

تقسیم بندی Flynn

در این روش کامپیوتر را از نظر تعداد دستورات و داده‌های دستکاری شده در یک زمان تقسیم بندی می‌کنند.

- رشته‌ی تک دستوری، رشته تک داده ای(SISD)
- رشته‌ی چند دستوری، رشته تک داده ای(SIMD)
- رشته‌ی تک دستوری، رشته تک داده ای(MISD)
- رشته‌ی تک دستوری، رشته تک داده ای(MIMD)

منظور از تک دستوری بود و تک داده‌ای بودن این است که یک سیستم کامپیوتری بتواند چند دستور را به طور موازی در یک زمان اجرا کند. در صورتی که یک کامپیوتر چند دستوری باشد نیاز به چند پردازش گر نیز خواهد داشت

منظور از چند داده‌ای بودن این است که یک سیستم بتواند یک یا چند دستور را به طور همزمان روی داده‌های متفاوت اجرا کند. برای مثال کامپیوتری که از نوع MISD باشد نیاز به چند پردازش گر دارد تا بتواند در یک زمان یک دستور را بر روی چند داده‌ی مختلف اجرا کند.

نوعی دیگر از تقسیم بندی پردازش موازی:

- **خط لوله** تکنیکی است که یک پردازش سری را به عملایت جزیی تفکیک می‌نماید و هر عمل جزیی در مقطع خاصی همزمان با سایر مقاطع اعمال می‌شود. در واقع خط لوله مجموعه از قطعه پردازش کننده‌ها است که هر کدام بخشی را که به آن دیکته شده است انجام می‌دهد.
- **پردازش برداری** درباره‌ی محاسبات بردارها
- **پردازش آرایه‌ای** هم بر روی آرایه‌های بزرگ عمل می‌کند.

ساختار خط لوله(نگاهی ساده):

هر قطعه از یک ثبات ورودی و بدنبال آن یک مدار ترکیبی ساخته شده است. ثبات داده را نگه می‌دارید و مدار ترکیبی جز عملی را در ارتباط با قطعه انجام می‌دهد.

خط لوله حسابی:

این نوع از خط لوله در کامپیوترهای بسیار سریع و برای پیاده سازی اعمال حسابی چون ضرب و تقسیم به کار می‌روند. برای مثال در یک جمع کننده ممیز شناور قطعات عبارتند از:

- مقایسه نماها
- همردیف کردن مانتیس ها
- جمع یا تفریق مانتیس ها
- نرمالیزه کردن نتیجه

خط لوله دستور العمل:

در یک خط لوله‌ی دستور العمل سعی شده است فازهای برداشت و اجرای دستورات بر هم منطبق و همزمان انجام گیرد.

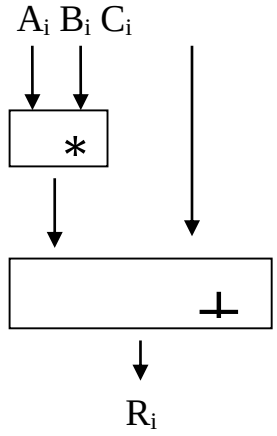
- Pipeline برای افزایش سرعت در کارهای ترتیبی به کار می‌رود.

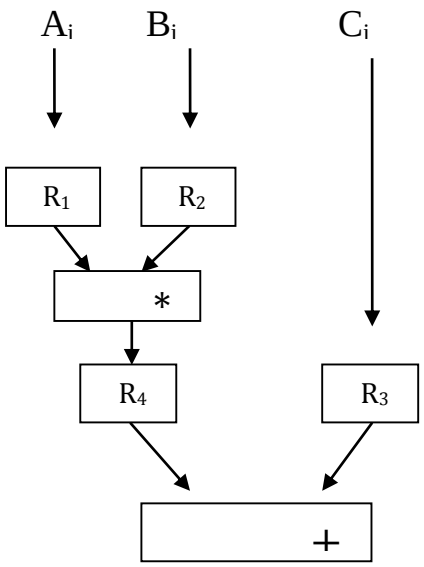
برای تفهیم بهتر خط لوله آن را از طی یک مثال بررسی می‌کنیم:

فرض کنید همیشه بخواهیم $A_i * B_i + C_i$ را روی داده‌های مختلف حساب کنیم که A, B, C سه عدد هستند که در ورودی داده می‌شوند. همچنین فرض می‌کنیم برای هر واحد t_{delay} زمان بخواهیم و پایه Clock همه ثبات‌ها یکسان است.

اگر بخواهیم این محاسبات را از روش معمولی محاسبه کنیم، برای هر محاسبه زمانی برابر با $۲t_{delay}$ نیاز خواهیم داشت و برای n محاسبه‌ی مختلف نیاز به $n*2*t_{delay}$ خواهیم داشت که زمان نسبتا بالایی خواهد بود.

برای استفاده از سازمان خط لوله طراحی را به صورت زیر تغییر می‌دهیم:





در این صورت پردازش‌ها در قطعه به صورت زیر است:

clock	R1	R2	R3	R4
1	A1	B1		
2	A2	B2	C1	A1 * B1
3	A3	B3	C2	A2 * B2
4			C3	A3 * B3

اگر دقت کنید بعد از دفعه اول بقیه‌ی محاسبات تنها به یک واحد زمانی تاخیر دارد. درنتیجه برای محاسبه‌ی n بار فرمول فوق نیاز به n+1 کلاک داریم تا کل محاسبات اتمام یابد.

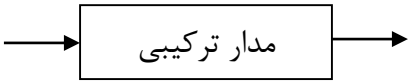
اگر مثلا تاخیر ضرب کننده T و تاخیر جمع کننده‌ی یک بیتی هم T باشد، در این صورت طول clock در حالت دوم باید:

$t_{clock} \geq (n+1)T \rightarrow t_{clock} = (n+1)T$

در حالت اول باید 2nT صبر کنیم. پس حالت اول نسبت به حالت دوم $\frac{2nT}{(n+1)T}$ یعنی تقریبا ۲ برابر کندتر است. به این تکنیک، خط لوله (Pipeline) می‌گویند.

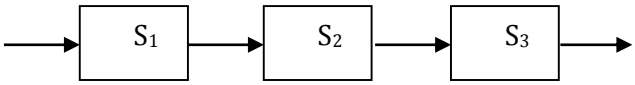
حالت کلی:

گفتیم هر مدار خط لوله از یک تعدادی مدار

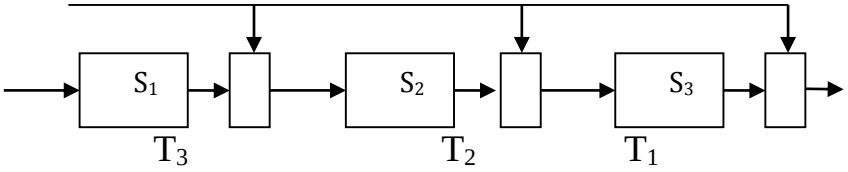


ترکیبی یا قطعه پردازش گر تقسیم شده است.

مدارهای ترکیبی را به گام[۲۷]هایی که جدا از هم هستند تقسیم می‌کنیم. مثلا:



در حالت کلی سیم بین این مدارها را قطع کرده و بین آنها رجیستر می‌گذاریم.



در مدار فوق هر کدام از قطعه‌ها کار خود را به ترتیب در T1,T2,T3 به اتمام می‌رسانند. از آن جایی که یک مدار خط لوله زمان واحدی بین قطعات پردازش گر آن باید موجود باشد تا به طور همزمان محاسبات هر قسمت به بخش قبلی منتقل شود باید طول کلاک را طوری تعیین کنیم که از هیچکدام کوچکتر نباشد پس داریم:

$$\text{طول کلاک} \geq \text{Max}(T_1, T_2, T_3)$$

تاخیر دفعه‌ی اول محاسبات برابر $T_1 + T_2 + T_3$ است.

برای سادگی فرض کنید T_iها مساوی و برابر t_n و تعداد stageها k باشد. پس

تاخیر انجام n تا task با مدار ترکیبی بدون خط لوله $n(kt_n)$ خواهد بود. این در حالی است که دفعات بعدی محاسبات تنها بعد از یک واحد تاخیر از مرحله قبل به اتمام می‌رسند پس داریم:

تاخیر انجام n تا task با خط لوله = تاخیر اولین task1 + $(n\text{-}1)\ t_{clock}$

برای محاسبه‌ی میزان افزایش سرعت زمان از فرمول زیر استفاده می‌کنیم:

---

۲۷ Stage

$$\text{Speed-up} = \frac{\text{کند}}{\text{تند}} = \frac{nkt_n}{kt_n+(n+1)t_n} = \frac{nk}{k+n-1}$$

حال اگر k ثابت باشد و اما n متغیر و اگر $n \to \infty$ میل کند خواهیدم داشت:

Speed-up=k

از محاسبات فوق به این نتیجه می‌رسیم که ماکسیمم افزایش سرعت با خط لوله به اندازه تعداد stageها است. در نتیجه هر چه stageها را بیشتر کنیم بهتر است. اما بعضی اوقات نمی‌توان stageها را کوچکتر کرد. مثلا یک گیت با دو ورودی که هر ورودی طولی متفاوت دارد. ضمن این که زیاد شدن k تعداد ثبات‌ها را زیاد می‌کند که در این صورت، overhead خیلی بیشتر از حالت اول می‌شود.

مثال: اگر سه stage باشد و هر کدام به ترتیب ۲، ۵ و ۱۰ نانوثانیه باشد و ۱۰۰ دستور اجرا کنیم، چقدر speed-up داریم؟

پاسخ:

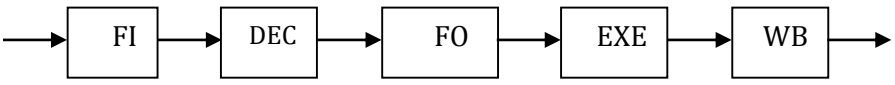
حالت عادی: (۲+۵+۱۰) × ۱۰۰ = ۱۷۰۰

حالت خط لوله: ۳۰ + ۹۹ × ۱۰ = ۱۰۲۰ (طول کلاک = ۱۰)

$$Speed\text{-}up = \frac{1700}{1020} \approx 1/6$$

گفتیم میزان زمان تاخیر هر مرحله باید به مقدار ماکسییم تاخیر هر کدام از قطعه‌ها باشد. پس گاهی اوقات تقسیم محسبات به قعات ریز تر می‌تواند کار ساز نباشد. برای مثال در بالا می‌توان ۲ و ۵ را یکی کرد. در واقع خط لوله‌ای خوب است که در آن تاخیر stageها به هم نزدیک باشد و الزاما تقسیم کردن بر اساس سخت افزار درست نیست بلکه باید بر اساس زمان باشد.

همین ایده را می‌توان در بخش کنترل کننده‌ی کامپیوتر هم ایجاد کرد. کار ترتیبی در کامپیوتر مطابق الگوریتم فون-نیومن است که می‌توان آن را به جز پردازش گرهای زیر تقسیم بندی کرد :



در تقسیم بندی فوق هر کدام از مراحل مختلف به طور موازی در یک زمان اما روی دستورات مختلف خوانده شده از حافظه اجرا می‌شود. به گونه‌ای که ابتدا در قطعه اول یک دستور خوانده شده و در زمان بعدی برای دیکد شدن به قطعه‌ی بعدی منتقل می‌شود و در همان زمان دستور بعدی از حافظه خوانده می‌شود.

بررسی یک مشکل در این روش:

فرض کنید در جدول زیر T_4، یک دستور jump از نوع conditional (شرطی) باشد. برای مثال T_4 $jump_z$ 100H باشد. در این صورت T_5 با توجه به مقدار zero flag دستور بعدی که باید اجرا شود یا دستوری است که در خانه‌ی 100H قرار دارد و یا دستوری است که در خط بعد از T_4 آمده است. پس T_5 باید صبر کند تا T_4 تمام شود و سپس شروع به کار کند. وقتی که دستور T_4، execute شود تکلیف zero flag مشخص میشود سپس میتوان T_5 شروع به کار کند.

در این موارد بین دو دستور اصطلاحاً حباب ایجاد می‌شود (که در شکل با هاشور نمایش داده شده است). به این اتفاق Stall می‌گویند. هر چه stall بیشتری اتفاق بیفتد به ضرر ماست. و بدبینانه ترین حالت برای pipeline زمانی است که فقط jumpهای شرطی داشته باشیم. با این حال می‌توان گفت pipeline در مجموع کارایی مناسبی دارد. [۲۸]

مکان / مرحله	۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱
FI	T1	T2	T3	T4					T5		
DEC		T1	T2	T3	T4					T5	
FO			T1	T2	T3	T4					T5
EXC				T1	T2	T3	T4				
WB					T1	T2	T3	T4			

اما این نوع jumpها فقط یکی از ضعف‌های روش pipeline است. به طور کلی می‌توان ۳ نوع نقطه ضعف برای این روش متصور شد که به آن اشاره می‌کنیم:

۲۸ آمار نشان میدهد در هر ۱۰۰ خط یک jump در برنامه‌ها وجود دارد.

نقاط ضعف روش pipeline

۱. وابستگی فیزیکی: فرض کنید در یک stage از عمل pipeline همزمان می‌خواهیم operand یک دستور را بدست آوریم که نسبی است (مثلا دستور add b_1[4] که در آن b_1 یک ثبات پایه است) و عمل execute یک دستور جمع را بدست آوریم. مشخصا در هر دو دستور به ALU نیاز داریم (در اولی برای جمع کردن b_1 و 4 و در دومی برای خود دستور جمع کردن) و این یک recourse conflict است.

برای حل این مشکل می‌توان در صورتی که می‌توانیم هزینه بیشتری بپردازیم چند ALU بخریم در غیر اینصورت Stall ایجاد کنیم.

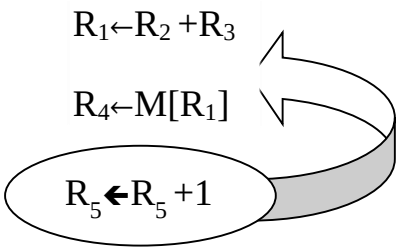
۲. وابستگی داده ای: دستورات رو به رو را در نظر بگیرید:

$R1 \leftarrow R2 + R3$

$R4 \leftarrow M[R1]$

بدیهتاً دستور دوم باید بعد از دستور اول اجرا شود یعنی تا انجام عمل WB دستور اول صبر کند.

یک از روش‌های مجود برای حل این مشکل می‌توان تا جایی که ممکن است از این وابستگی‌ها در کدها پرهیز کنیم یعنی برای مثال compiler را ملزم به جلوگیری از وقوع این مشکلات کنیم. برای مثال کد زیر را در نظر بگیرید.



می توان با جا به جا کردن خط سوم بین خط اول و دوم مشکل وابستگی داده‌ای را حل کرد. به این کار تاخیر ایجاد کردن می‌گویند.

روش دیگر استفاده از سخت افزارها یی است که با جهت دهی به داده در مسیرهای خاص مشک را حل می‌کنند. برای مثال چک می‌کند اگر نتیجه محاسبات ALU در دستور بعدی مورد

نیاز است به جای انتقال آن به رجیستر ابتدا آن را به عنوان ورودی به واحد مورد نیاز می‌فرستد.

یکی دیگر از روش‌های استفاده از مدارهای همبند سخت افزاری (Hardware Interlock) می‌باشد که در این روش دستوراتی که ورودی‌های آن‌ها به خروجی‌های دستو رات قبلی نیاز دارد و منابع آن‌ها در دسترس نیست را شناسایی می‌کند و آن‌ها را با تاخیر اندکی اجرا می‌نماید.

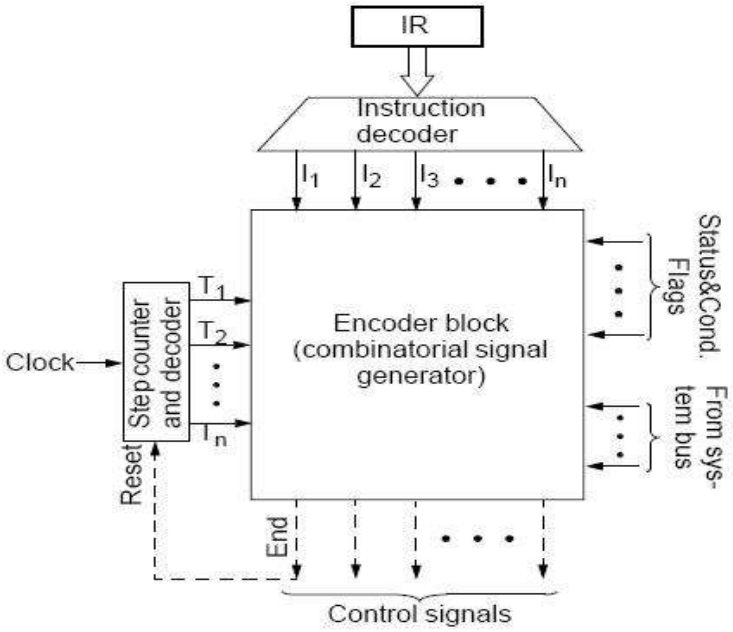
۳. وابستگی مربوط به branch ها: که به طور مفصل در conditional jump به آن اشاره شد.

از بین این سه مشکل branch problem قابل حل نیست اما می‌توان روش‌هایی را برای کمتر شدن تاخیر به کار برد. برای مثال می‌توان از branch prediction استفاده کرد یعنی پیش بینی کنیم که jump کنیم یا خیر. بعد از مشخص شدن نتیجه شرط اگر درست عمل نکرده باشیم باید همه‌ی کارهای اضافه انجام شده را دور بریزیم.

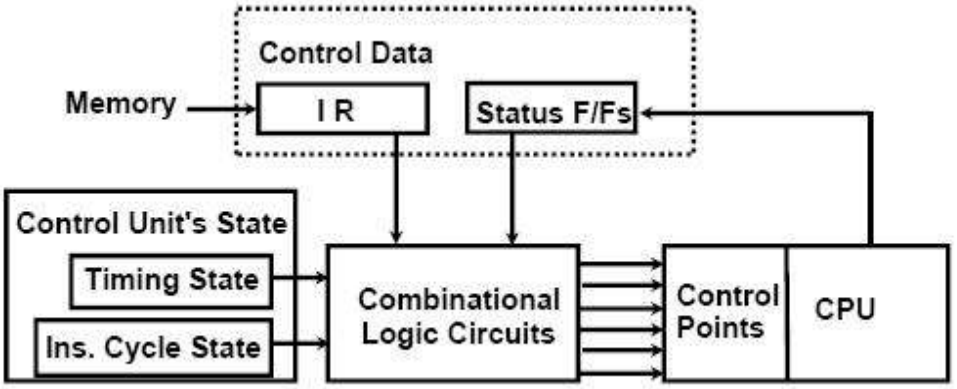
برای بهبود این روش می‌توان با نگه داری اطلاعاتی خاص، در مرحله branch prediction بهتر عمل کنیم بدین صورت که از یک جدول Branch Target Buffer (BTB) استفاده میکنیم و هر بار که به یک دستور jump برسیم آن را در این جدول قرار می‌دهیم و مشخص می‌کنیم به کدام آدرس jump کرده و بدین صورت در jumpهای بعدی می‌توانیم آن آدرسی را برای jump کردن انتخاب کنیم که در دفعات قبل بیشتر به آن jump شده است. از انجا که BTB حافظه محدود دارد فقط تاریخچه‌ی کوتاهی از jumpها را نگه می‌دارد.

واحد کنترل– سیم بندی شده (Hard-wired):

در این حالت، واحد کنترل مداری ترکیبی است که مجموعه‌ای از ورودی‌ها (شامل IR، flag، clock) را به مجموعه‌ای از سیگنال‌های کنترلی تبدیل می‌کند.



شکل ۷



شکل ۸

ویژگی‌ها:

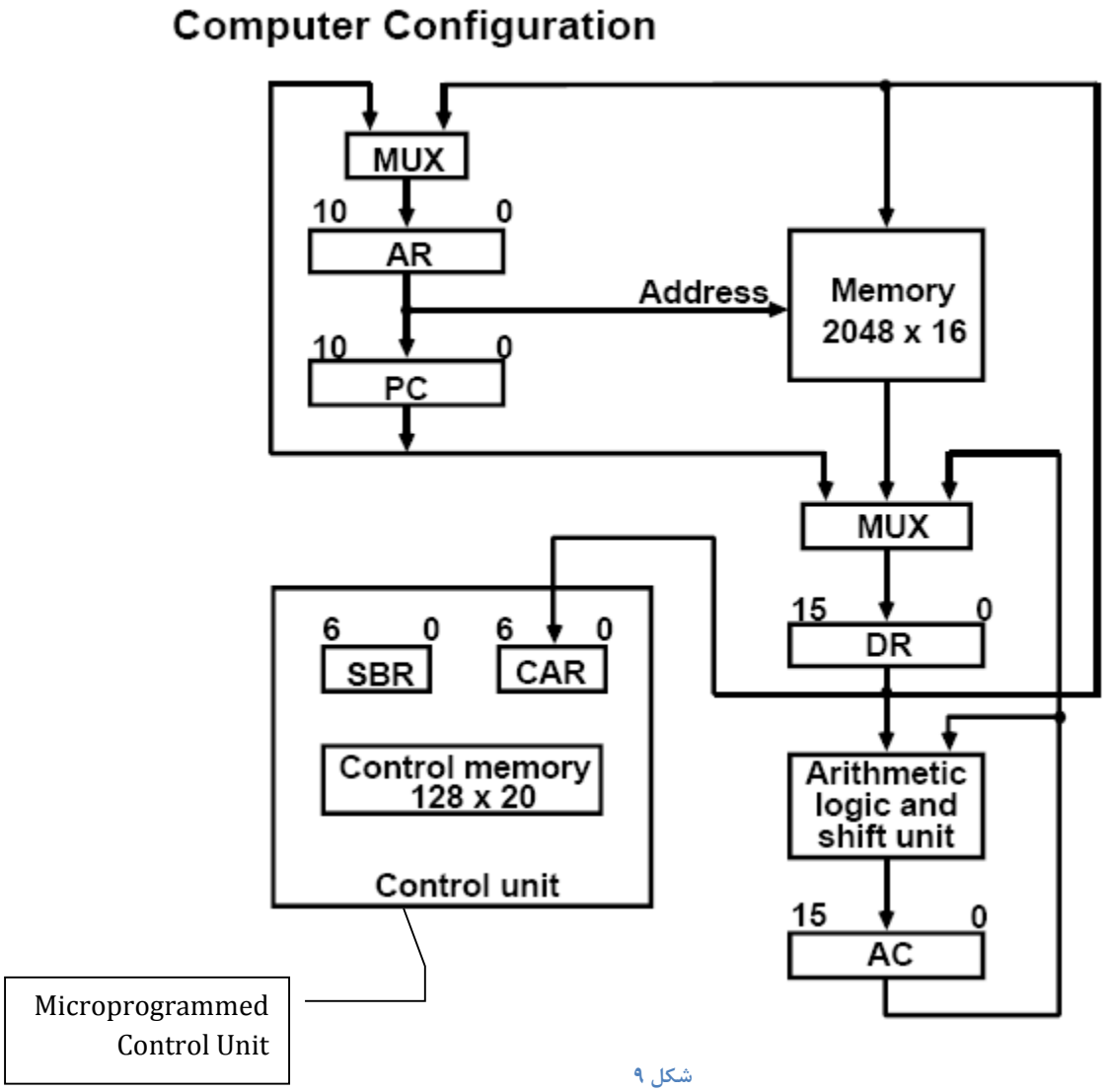
- سرعت بیشتر در مقایسه با واحد کنترل برنامه‌پذیر
- در صورت پیچیدگی دستورالعمل‌ها (در معماری CISC) پیاده‌سازی واحد کنترل سیم بندی شده مشکل خواهد بود، در این حالت از واحد کنترل برنامه‌پذیر استفاده می‌شود.

- دستورالعمل‌ها ثابت و غیرقابل تغییر هستند (درحالیکه واحد کنترل برنامه‌پذیر قابلیت تغییر عملکرد دستورالعمل‌ها را دارد.)

کنترل برنامه پذیر (Microprogrammed Control Unit):

بجای مدار ترکیبی یک حافظه برنامه پذیر وجود دارد که عملیات کنترلی بوسیله ریزدستورالعمل‌هایی که در آن است انجام می‌شود.

در واقع ریزدستورالعمل‌ها از مجموعه‌ای از ریزعملگرها تشکیل شده است که برای اجرای ریزعملوندها از مدارات سیم بندی شده استفاده می‌شود.



شکل ۹

ریزبرنامه (Microprogram):

مجموعه‌ای از ریزدستورالعمل‌ها که در حافظه برنامه‌پذیر ذخیره شده و سیگنال‌های کنترلی موردنیاز برای اجرای مجموعه دستورالعمل‌ها را تولید می‌کند.

ریزدستورالعمل (Microinstruction):

از دو قسمت تشکیل شده است:

کلمه کنترلی (Control Word) که شامل همه‌ی اطلاعات مورد نیاز برای یک کلاک

کلمه ترتیب‌دهی (Sequencing Word): اطلاعات موردنیاز برای تشخیص ریزدستور بعدی

حافظه کنترلی (Control Storage: CS):

حافظه‌برای ذخیره ریزبرنامه‌ها

حافظه کنترلی نوشتنی (Writable Control Storage):

حافظه برنامه‌پذیری است که محتویاتش که همان مجموعه دستورالعمل‌ها است قابل تغییر هستند.

ریزبرنامه‌نویسی پویا (Dynamic Microprogramming):

کامپیوتری که حافظه کنترلی آن از نوع نوشتنی است و ریزبرنامه آن قابلیت تغییر توسط کاربر را دارد.

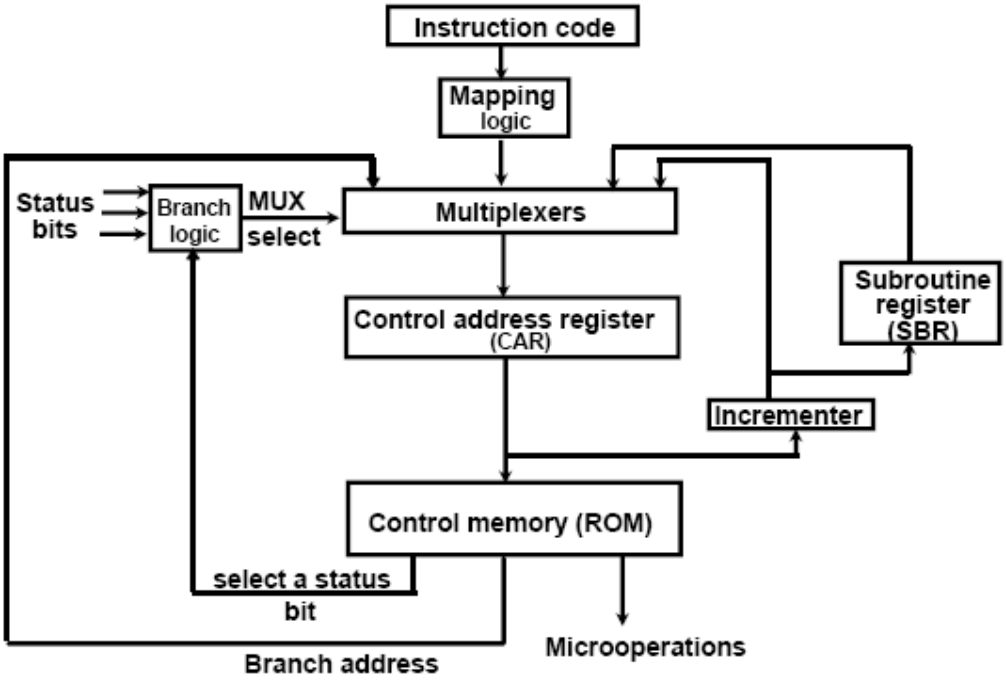
توالی‌گر (Sequencer):

آدرس ریزدستورالعملی که در کلاک بعدی باید اجرا شود را تعیین می‌کند.

انواع توالی‌ها:

- In-line Sequencing: رفتن به دستورالعمل بعدی
- Branch: به دستورالعمل با آدرسی که در فیلد آدرس دستو العمل فعلی آمده پرش می‌کند.
- Conditional Branch: اگر شرط مورد نظر درست بود، به آدرس اشاره شده پرش می‌کند.
- Subroutine: پرش به آدرس اول زیرروال مورد نظر.
- Loop

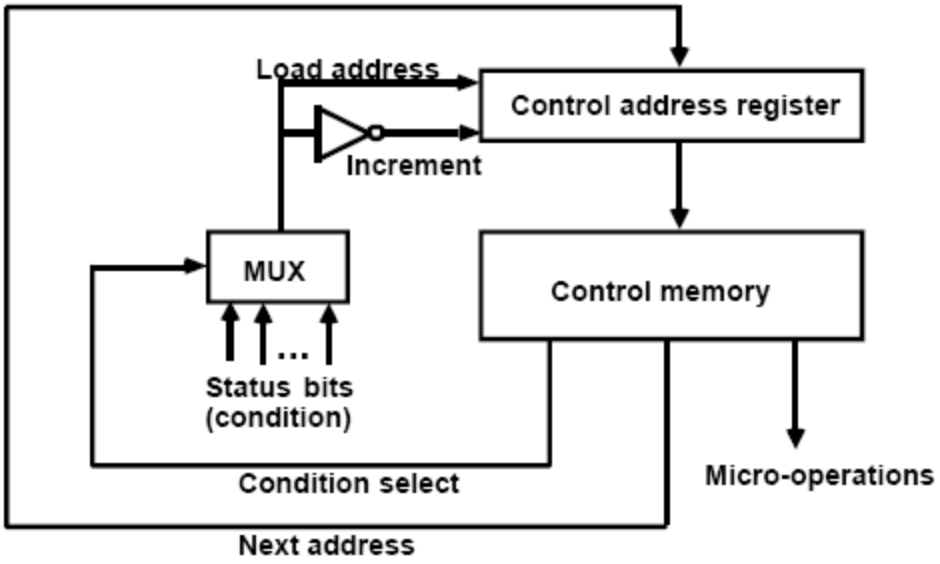
توالی ریزدستورالعمل‌ها



شکل ۱۰

ابتدا دستورالعمل از روی حافظه برروی قسمت (۱) قرار می‌گیرد و توسط قسمت (۲) آدرس شروع روال مربوط به آن دستورالعمل مشخص می‌شود و در CAR قرار داده می‌شود. محتویات متناظر با این آدرس که یک ریزدستورالعمل است از حافظه کنترلی خوانده می‌شود و ریزعملگرهای اجرا می‌شوند (F1، F2، F3)؛ سپس بیت‌های وضعیت به (۳) منتقل شده و نوع توالی آن مشخص شده و به مالتی‌پلکسر رفته و مالتی پلکسر آدرس ریزدستورالعمل بعدی را از میان ۴ ورودی آن انتخاب می‌کند. ورودی‌ها شامل آدرس انشعاب[۲۹]، آدرس شروع زیرروال، آدرس ریزدستورالعمل بعدی، آدرس روال مربوط به هر دستورالعمل (که در قسمت (۲) مشخص می‌شود.)

۲۹ Branch Address



شکل ۱۱

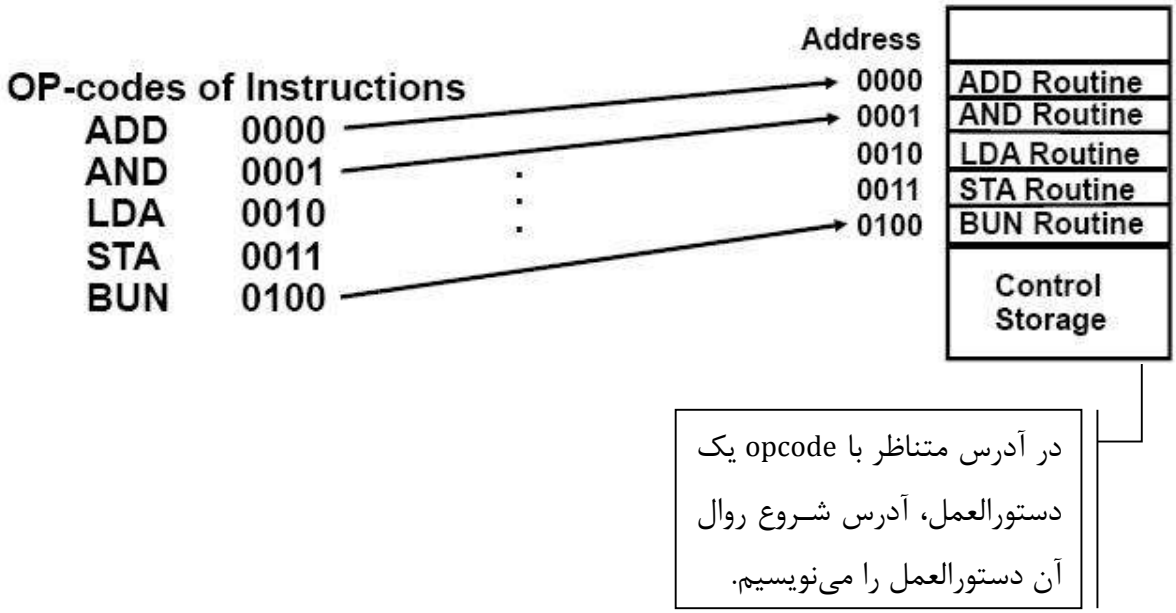
به عنوان مثال در حالت انشعاب شرطی، نوع شرط (carry, zero, sign, overflow و. ..) در مالتی پلکسر توسط condition select انتخاب می‌شود و مقدار آن شرط در خروجی مالتی پلکسر می‌رود و در صورت درست بودن، آدرس محل پرش در CAR قرار می‌گیرد و ادامه ریز دستورالعمل‌ها از این آدرس اجرا می‌شود و اگر غلط بود، CAR را یک واحد اضافه می‌کند و به ریز دستورالعمل بعدی می‌رود.

یکی از بیت‌های وضعیت همواره ۱ است و در حالت انشعاب غیر شرطی، condition select این بیت را از ورودی مالتی پلکسر انتخاب می‌کند. بنابراین شرط پرش همواره درست است.

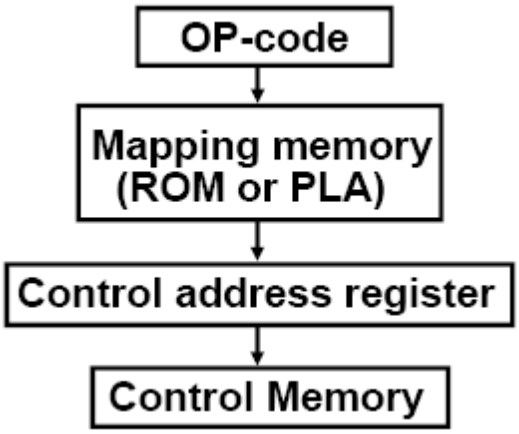
نگاشت دستورالعمل‌ها:

در این قسمت opcode هر دستورالعمل به آدرس شروع روال مربوط به آن در حافظه کنترلی نگاشته می‌شود.

یکی از روش‌های آن نگاشت مستقیم است که پیاده‌سازی آن به وسیله ROM یا PLA صورت می‌گیرد.



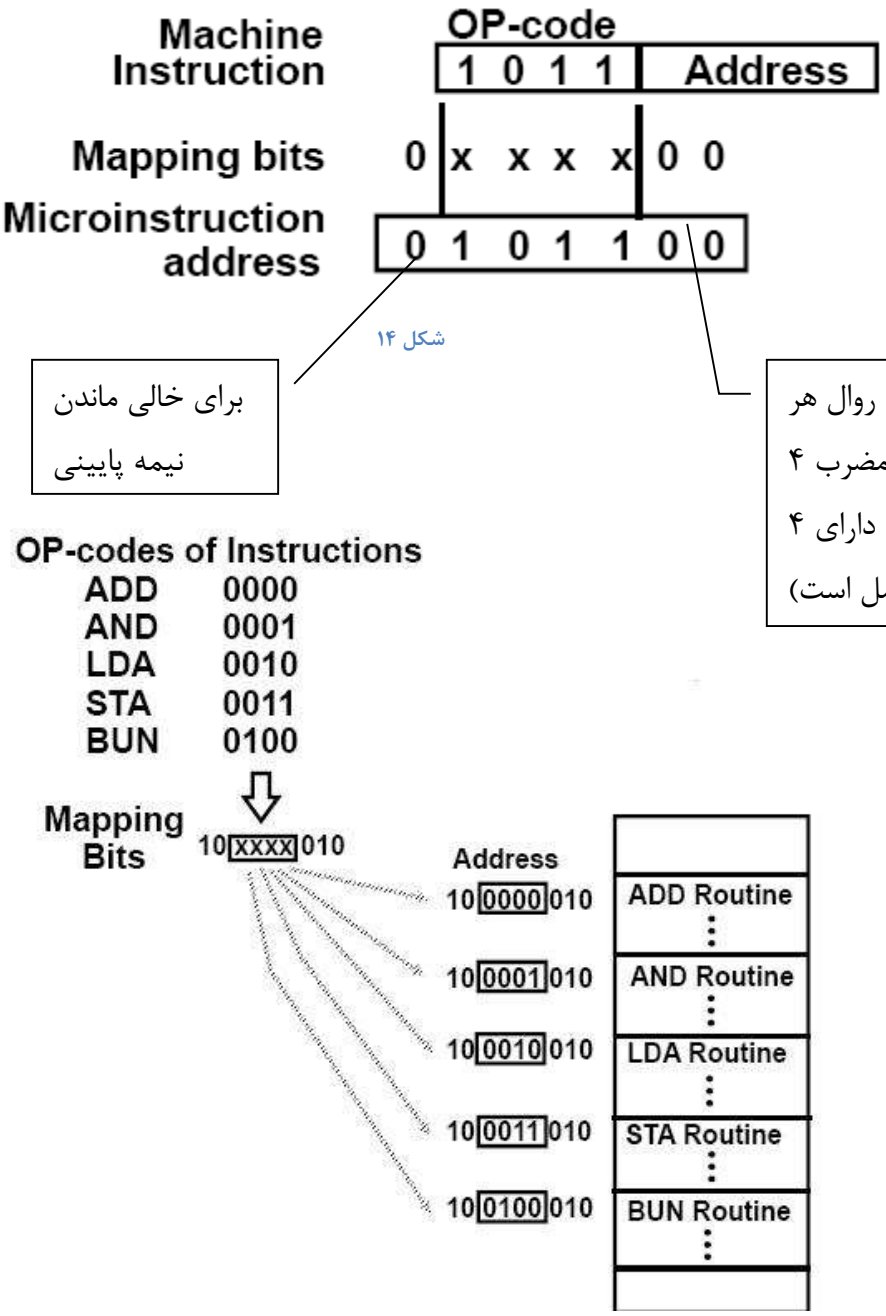
شکل ۱۲



شکل ۱۳

روش دیگر، نگاشت بیتی است: در این روش به ابتدا و انتهای opcode یک دستورالعمل تعدادی بیت اضافه می‌کنیم.

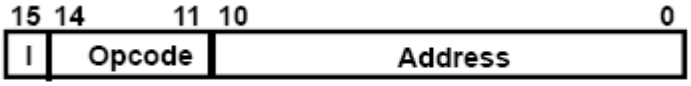
مثال:



شکل ۱۴

شکل ۱۵

قالب دستورالعمل ماشین:



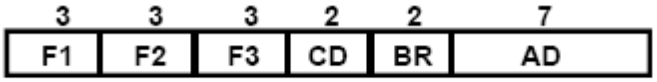
شکل ۱۶

نمونه‌ای از دستورالعمل ماشین:

Symbol	OP-code	Description
ADD	0000	AC ← AC + M[EA]
BRANCH	0001	if (AC < 0) then (PC ← EA)
STORE	0010	M[EA] ← AC
EXCHANGE	0011	AC ← M[EA], M[EA] ← AC

شکل ۱۷

قالب ریزدستورالعمل:



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

شکل ۱۸

فیلدهای ریزعملگر

F1,F2,F3: همان طور که در شکل بالا مشاهده می‌شود هر فیلد شامل تعدادی ریزعملگر است. در هر ریزدستور تمام ریزعملگرها همزمان اجرا می‌شوند و بعد انشعاب انجام می‌شود. هر کدام از فیلدها می‌تواند NOP باشد. ریزعملگرها باید طوری بین فیلدها تقسیم شوند که ریزعملگرهایی که لازم است همزمان اجرا شوند در فیادهای مختلف باشند تا از ریزدستورالعمل‌های کمتری اسفاده شود.

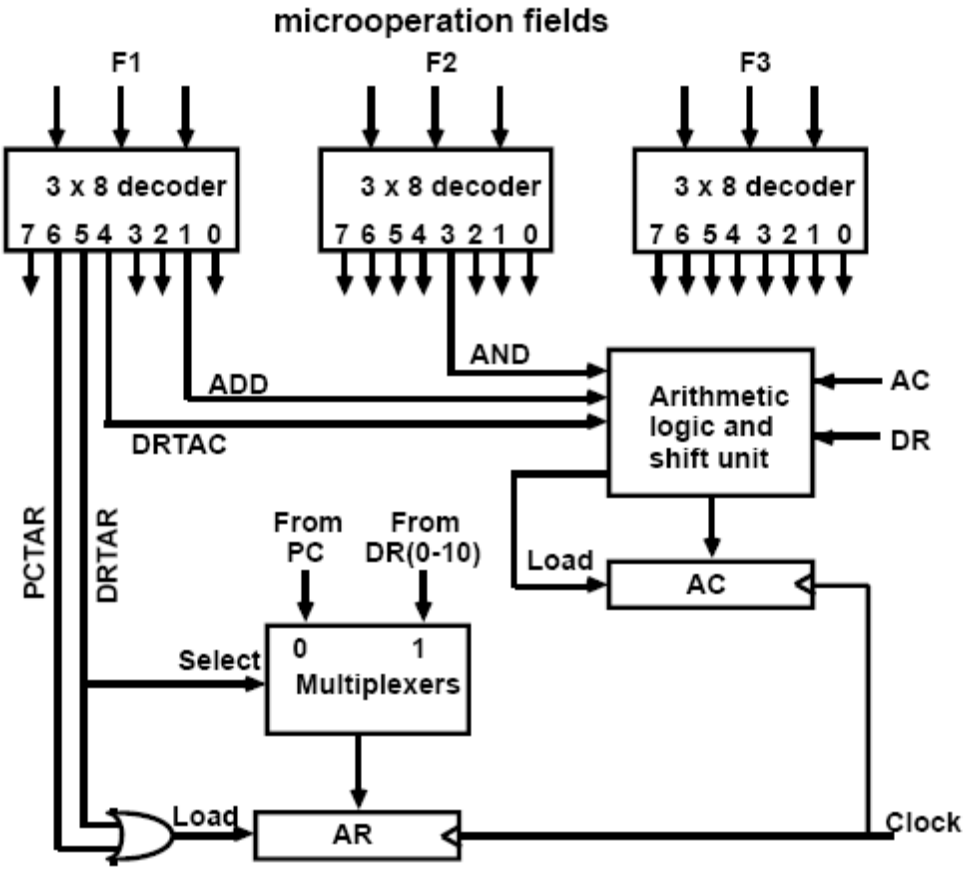
F1	Microoperation	Symbol
000	None	NOP
001	AC ← AC + DR	ADD
010	AC ← 0	CLRAC
011	AC ← AC + 1	INCAC
100	AC ← DR	DRTAC
101	AR ← DR(0-10)	DRTAR
110	AR ← PC	PCTAR
111	M[AR] ← DR	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	AC ← AC - DR	SUB
010	AC ← AC ∨ DR	OR
011	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR + 1	INCDR
111	DR(0-10) ← PC	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	AC ← AC ⊕ DR	XOR
010	AC ← AC'	COM
011	AC ← shl AC	SHL
100	AC ← shr AC	SHR
101	PC ← PC + 1	INCPC
110	PC ← AR	ARTPC
111	Reserved	

شکل ۱۹

نحوه‌ی اجرای برخی از ریزدستورالعمل‌ها در واحد کنترل به صورت زیر است. همان طور که مشاهده می‌شود اجرای ریزدستورالعمل‌ها با استفاده از مدارات ترکیبی و به صورت hard-wired است.



شکل ۲۰

CD: این فیلد همیشه دو بیت است و برای تشخیص نوع شرط‌ها به صورت زیر اینکد می‌شود اگر این مقدار برای ۰۰ باشد بدان معنا است که این شرط همواره بر قرار است و استفاده از این شرط همراه با میدان BR یک انشعاب بدون شرط را می‌سازد.

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

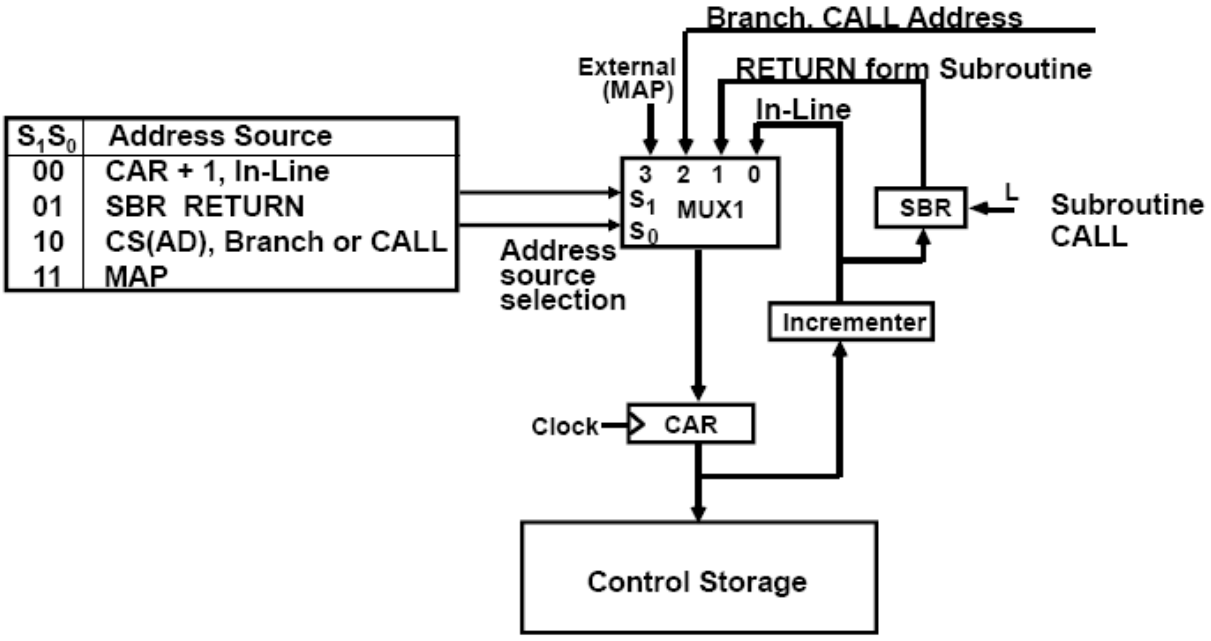
شکل ۲۱

BR: این قالب نوع شرط را نشان می‌دهد. این میدان هم مانند CD ۲ بیت است و به صورت زیر انکد شده و انواع شرط را نشان می‌دهد. قابل ذکر است که دو حالت ۰۰ و ۰۱ تقریبان یکسان هستند با این تفاوت که ریز عمل CALL آدرس بازگشت را در رجیستر SBR ذخیره می‌کند.

BR	Symbol	Function
00	JMP	CAR ← AD if condition = 1
		CAR ← CAR + 1 if condition = 0
01	CALL	CAR ← AD, SBR ← CAR + 1 if condition = 1
		CAR ← CAR + 1 if condition = 0
10	RET	CAR ← SBR (Return from subroutine)
11	MAP	CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

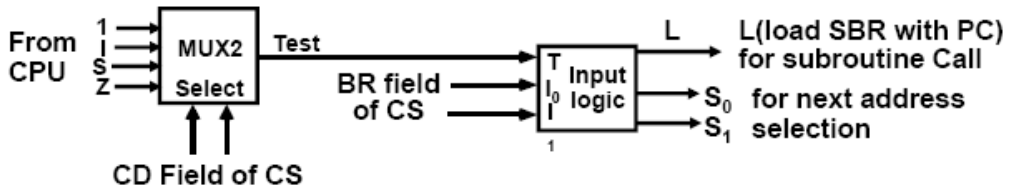
شکل ۲۲

نحوه‌ی تشخیص توالی اجرای ریزدستورالعمل ها:



شکل ۲۳

S_1,S_0خروجی مدار زیر است که با استفاده از BR و CD مشخص می‌شود.



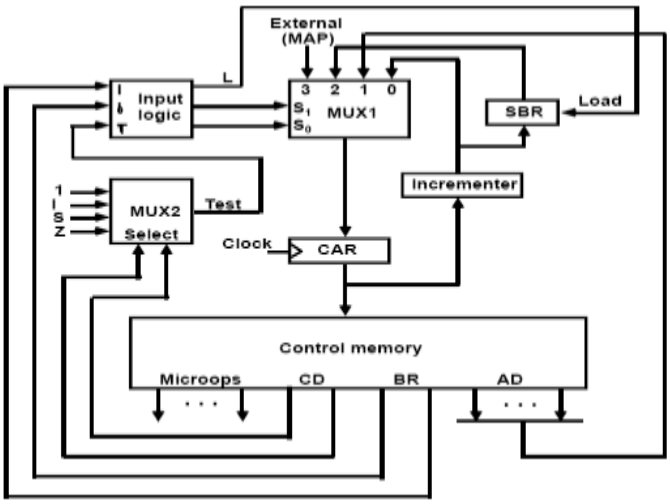
Input Logic

I_0I_1T	Meaning	Source of Address	S_1S_0	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	10	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	10	1
10x	RET	SBR	01	0
11x	MAP	DR(11-14)	11	0

$$S_0 = I_0$$
$$S_1 = I_0I_1 + I_0'T$$
$$L = I_0'I_1T$$

شکل ۲۴

که در کل بخش تعیین کننده ترتیب اجرای ریزدستورالعمل‌ها به صورت زیر است:



شکل ۲۵

ریز دستورالعمل‌های نمادین

در ریز دستورالعمل‌ها نیز همانند زبان اسمبلی از نمادها استفاده می‌شود. که در این صورت توسط یک اسمبلر به معادل دودویی آن تبدیل می‌شود.

هر قالب از پنج فیلد تشکیل شده است که نمایش نمادین آن بصورت زیر است

- Label همان آدرس نمادین است که باید به':' ختم شود و صرفا آدرس شروع یک‌سری از دستورالعمل‌ها را مشخص می‌نماید و می‌تواند وجود نداشته باشد.
- Micro-ops شامل یک، دو یا سه نماد است که هر کدام از نمادها نماینده یکی از ریزعملگرها است (که در بالا به آنها اشاره شد) که با کاما جدا می‌شوند.
- CD یکی از نمادهای U, I ,S,Z است. (شکل ۱۵)
  - U:انشعاب غیرشرطی
  - I: بیت مشخص کننده آدرس‌دهی غیرمستقیم
  - S: علامت AC

- Z: صفر بودن AC

- BR یکی از ۴ نماد JMP، CALL، RET، و MAP (شکل ۱۶)

- AD یکی از حالت‌های:

  - آدرس نمادین (label)

  - NEXT (ریزدستورالعمل بعدی)

  - ویا هیچی ! (هنگامیکه BR مشخص کننده آدرس ریزدستورالعمل بعدی است مثل MAP و RET)

مثال:در این مثال حافظه کنترلی شامل ۱۲۸ کلمه ۲۰ بیتی است که ۶۴ کلمه اول آن برای روال‌های ۱۶ دستورالعمل ماشین است و ۶۴ کلمه بعدی برای مقاصد خاص (!) مثل زیرروال‌ها استفاده می‌شود (که در این مثال در این بخش زیرروال fetch تعریف شده است)

نگاشت مورد استفاده همان نگاشتی است که در شکل ۸ بررسی شد.

در شکل ۲۰ نمایش نمادین ریزبرنامه‌ای که در این حافظه ذخیره شده است مشاهده می‌شود.

Label	Microops	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 64			
FETCH:	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

شکل ۲۰

معادل دودویی آن به صورت زیر است.

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

شکل ۲۱

قالب افقی و عمودی ریزدستورالعمل‌ها

قالب افقی: طول کلمه‌ها در این قالب زیاد است و همزمان و بصورت موازی قادر به کنترل و اجرای ریزعملگرهای یک ریزدستورالعمل است. در این‌صورت حافظه زیادی استفاده می‌شود.

قالب عمودی: به علت کدکردن فیلدهای ریزدستورالعمل‌ها طول کلمه‌ها در این قالب کم است ولی کدگشایی در این حالت به چندین مرحله نیاز دارد. همچنین در هر زمان تنها قادر به اجرای یک ریزعملگر خواهد بود.

نانوحافظه و نانو دستورالعمل

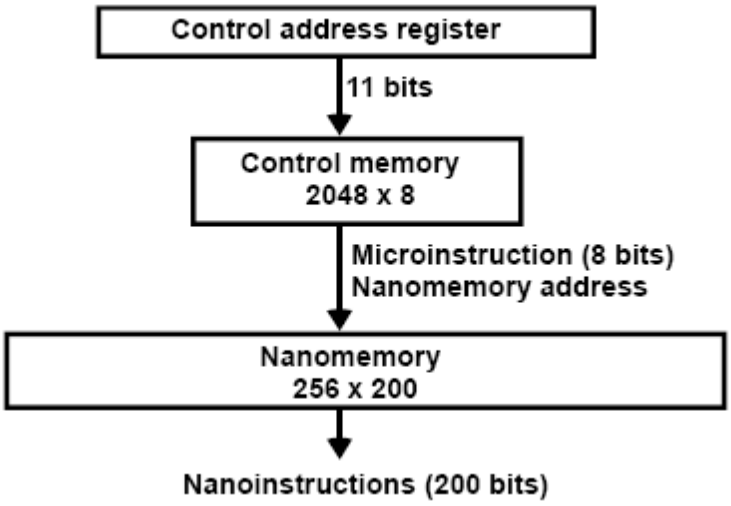
ریزبرنامه دو مرحله‌ای از دو سطح حافظه تشکیل شده است:

- سطح اول (حافظه کنترلی) که شامل ریزدستورالعمل‌های عمودی است.
- سطح دوم (نانوحافظه) از نوع افقی است که به دستورالعمل‌های آن نانودستورالعمل و به مجموعه‌ای از نانودستورالعمل‌ها، نانوبرنامه می‌گویند. برای تفسیر هر ریزدستورالعمل که در سطح اول قرار دارد، آن به یک نانودستورالعمل در سطح دوم تبدیل می‌شود.

معمولا ریزبرنامه شامل تعداد زیادی از ریزدستورالعمل کوتاه است، درحالیکه نانوبرنامه از تعداد کمتری نانودستورالعمل بلندترتشکیل شده است.

مثال:ریزبرنامه با ۲۰۴۸ ریزدستورالعمل ۲۰۰ بیتی، فرض کنیم از این ۲۰۴۸ ریزدستورالعمل، ۲۵۶ ریزدستور متمایز داریم.

- اگر از حافظه کنترلی یک سطحی استفاده استفاده کنیم، حافظه مورد نیاز ۲۰۴۸×۲۰۰=۴۰۹۶۰۰ بیت است.
- اگر از حافظه کنترلی دو سطحی استفاده استفاده کنیم،
  - نانو حافظه: شامل ۲۰۴۸×۲۰۰ بیت برای نگهداری ۲۵۶ نانودستورالعمل متمایز
  - حافظه کنترلی: شامل ۲۰۴۸×۸ بیت است چون برای آدرس دهی ۲۵۶ خانه‌ی نانوحافظه به ۸ بیت نیاز است. ←۲۵۶×۲۰۰+۲۰۴۸×۸=۶۷۵۸۴ بیت

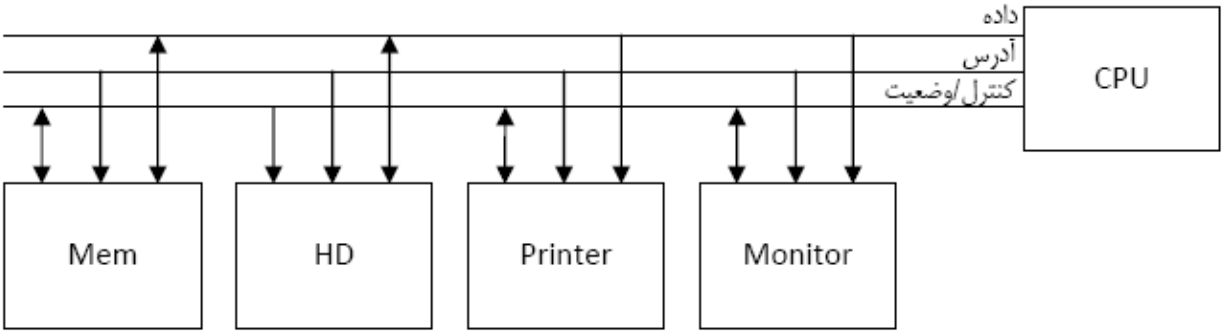


شکل ۲۶

برای اجرای یک ریزدستورالعمل، نوع ریزدستورالعمل در حافظه کنترلی تشخیص داده می‌شود و از نانوحافظه، نانودستورالعمل متناظر اجرا می‌شود.

ورودی / خروجی

I/O: دو چیز مد نظر است، چگونه و با چه پروتکلی باید به آن (ورودی و خروجی) دسترسی پیدا کنیم.

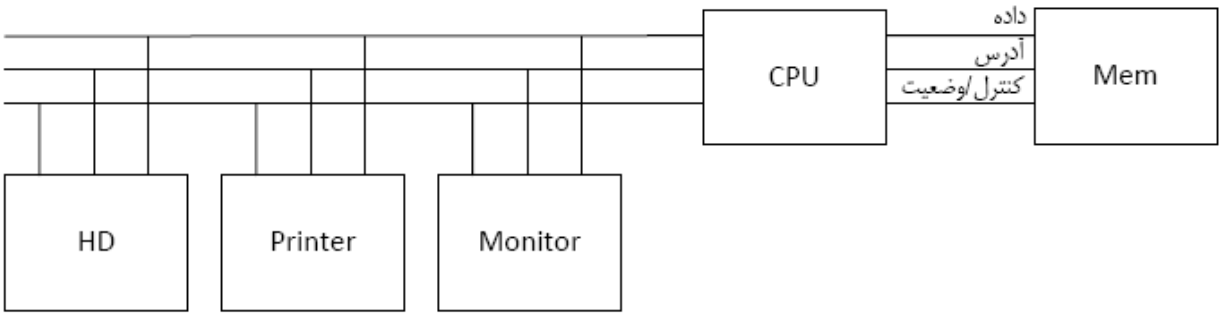


برای برقرار ارتباط می‌توان از سه روش زیر استفاده کرد:

۱. از busهای داده، آدرس و کنترل بطور مشترک استفاده شود، در این روش مشکلی وجود دارد، چرا که ممکن است بخواهیم همزمان از دستگاه‌ها استفاده کنیم. برای حل این مشکل می‌توان راه حل زیر را اعمال کرد:

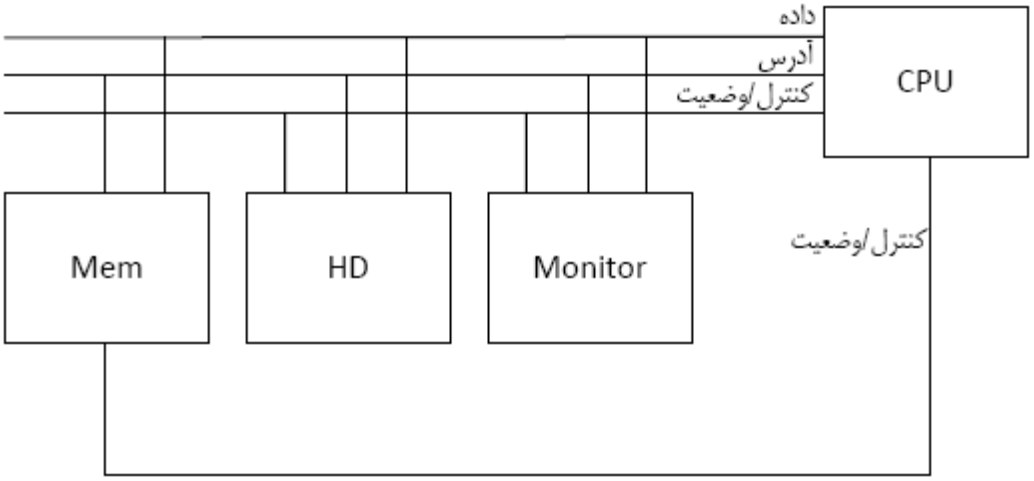
- *از آدرس برای تمایز بین دستگاه‌ها استفاده کنیم. مثلاً سه خانه حافظه را جدا کنیم و بگوییم هر گاه آدرس دهی به آن سه خانه بود، بسته به محل آن منظور دستگاههای جانبی یا حافظه است.*
- *ضعف این روش در این است که مجبوریم از حافظه فضایی را هدر دهیم. اصطلاحاً در این روش دستگاههای I/O به حافظه نگاشت شده‌اند. به این روش Memory Mapped I/O می‌گویند. حسن این روش سیم‌کشی خیلی کم است.*

۲. Bus داده، کنترل/وضعیت برای I/Oها جداگانه و bus حافظه نیز جداگانه متصل شود یعنی:



حسن این روش این است که دیگر داده و آدرس اشتباه نمی‌شوند، اما ضعف این روش در دو برابر شدن سیم-کشی‌ها و متعاقباً پیچیده شدن طراحی CPU است.

۳. از bus داده و آدرس مشترک استفاده شود ولی bus کنترل/وضعیت جداگانه متصل شود یعنی:



با این شیوه چون سیگنال کنترل جداست، مشکل اشتباه شدن داده پیش نمی‌آید. به این روش Isolated-I/O می‌گویند. در حال حاضر همین تکنولوژی استفاده می‌شود. البته باید توجه داشته که فقط ۲ سیگنال کنترل نیست بلکه سیگنالهای ready, write, read برای همه مشترک است وفقط یک سیگنال IO/M اضافه می‌کنند که اگر ۱ بود یعنی IO مد نظر است و اگر 0 بود یعنی Mem مورد نظر بوده است.

این برای سادگی و حداقل استفاده کردن سیم‌کشی‌ها استفاده می‌شود.

حال تصور کنید که از مورد سوم استفاده کنیم. در این حالت باید دید که چطور باید ارتباط را برقرار کرد. معمولاً از دستورات OUT و IN استفاده می‌کنیم که به عنوان ۱ اپرند شماره PORT واحد I/O داده می‌شود. این شماره‌ی PORT عملاً روی BUS آدرس قرار می‌گیرد و با این دستورات $IO/M = 1$.

برای ارتباط حافظه کافی است $IO/M = 0$ که در این حالت دستورات LOAD و STORE را داریم.

دستگاه‌های I/O کم است و به همین دلیل آدرس آنها کوچک است، تکنیک استفاده این است که از روی آدرس سیگنال IO/M، Chip Enable مربوط به دستگاه جانبی را فعال کنیم (با استفاده از and وor)

در این توپولوژی همیشه یک طرف CPU و یک طرف واحد I/O است. بعبارتی اگر مثلاً بخواهیم بین دو دستگاه مثلاً Mem و printer ارتباط براقرار کنیم باید اطلاعات ر اول به CPU ببریم و سپس از آنجا انتقال دهیم (بایت

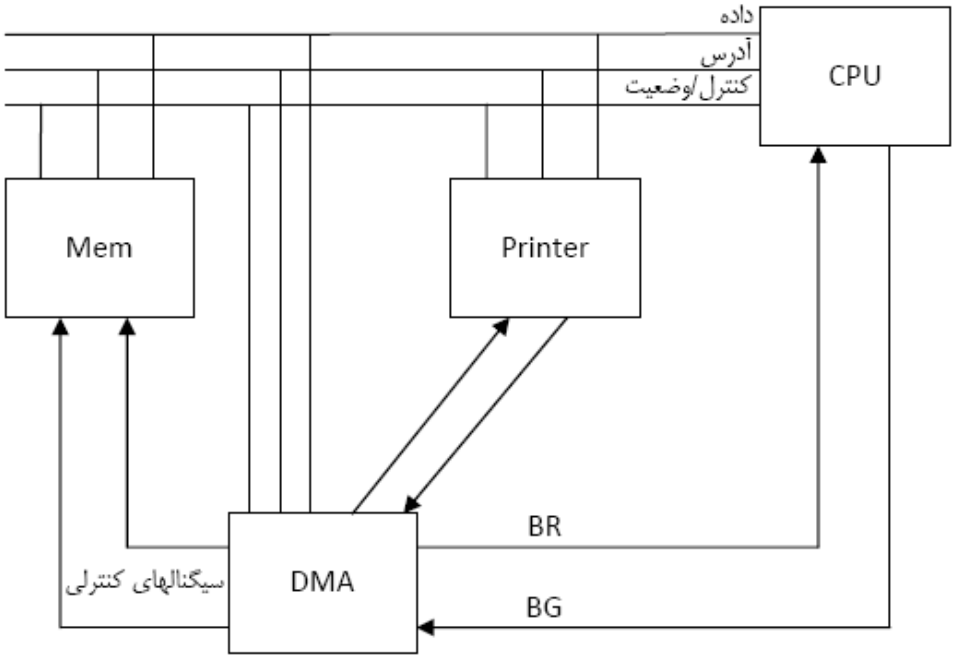
به بایت) اما مثلاً اگر بخواهیم یک فایل با حجم زیاد را انتقال دهیم سرعت کمی خواهیم داشت و CPU معطل خواهد شد. برای حل مشکل این پیشنهاد داده شد که در مواقع انتقال انبوه اطلاعات با کمترین زمان از واحد دیگری به نام DMA (Direct Memory Access) استفاده کنیم و این گونه بود که واحد DMA مطرح شد. (برای انتقال انبوه اطلاعات بین I/O و Mem همیشه یک طرف انتقال Mem است)

DMA دو سیگنال دارد:

- BR (Bus Request درخواست باس)
- BG (Bus Grant مجوز باس)

در مواقع انبوه اطلاعات از این دو سیگنال استفاده می‌کند، عملاً:

- ✓ هر وقت واحد I/O بخواهد با Mem انتقال انبوه انجام دهد به DMA درخواست می‌دهد. DMA به CPU درخواست می‌دهد (با BR) در صورتیکه CPU کارش با bus تمام شد (مثلاً وقتی که دستورالعمل تمام شد) از طریق BG مجوز استفاده از bus را به DMA می‌دهد.
- ✓ به محض گرفتن BG توسط DMA سیگنال ACK به واحد I/O داده میشود که می‌تواند مستقیماً به Mem ارتباط برقرار کند. در این حین CPU، Busهای داده، آدرس و کنترل را High z می‌کند.



در این صورت DMA مثل یک CPU انتزاعی است با این تفاوت که دیگر اطلاعات داخل DMA نمی‌آید، مثلاً همزمان به Mem میگوید اطلاعات را بخوان و در Data Bus بگذار و مثلاً به Printer می‌گوید write کن یعنی ارتباط Mem و Printer مستقیما با Data Bus برقرار شود.

DMA از Bus کنترل و آدرس برای مشخص کردن آدرس Mem و .. استفاده می‌کند. در ابتدای کار هم CPU آدرس مربوط به Mem را از طریق Bus داده به DMA میفرستد و در DMA در یک رجیستر ذخیره می‌شود.

اما اشکال کار در کجاست؟!

وقتی DMA کار می‌کند، CPU در اصطلاح talk نمی‌کند و کارایی بدی خواهد داشت و عملاً برنامه‌ی در حال اجرا متوقف شده و CPU کار نمیکند. یعنی برنامه متوقف شده و کارایی پایین آمده است، در واقع باید گفت که دو روش وجود دارد:

۱. Burst Transfer (انتقال انبوه)
۲. Stealing Cycle (سیکل دزدی)

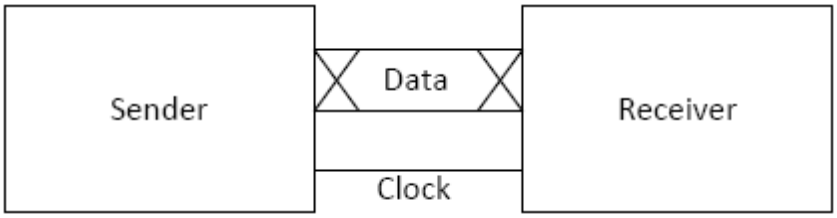
روش اول که در بالا توضیح داده شد و اما در روش دوم سیکلها یکی در میان به CPU و DMA داده می‌شود. یعنی یک بایت DMA انتقال می‌دهد و یک سیکل CPU کار می‌کند و. ..

به این ترتیب زمان انتقال اطلاعات بیشتری می‌شود اما کارایی پردازنده بهتر می‌شود اما مشکل اینجاست که CPU فکر کند که در Mem چیزی نوشته شود درحالیکه هنوز در حال انتقال اطلاعاتیم. یکی از راه حلها این است که می‌توان کاری کرد که CPU به آنجا آدرس دهی نکند و برنامه را متوقف نکند.

شیوه‌ی انتقال اطلاعات:

یعنی با چه شیوه‌ای و با چه پروتکلی اطلاعات انتقال یابد

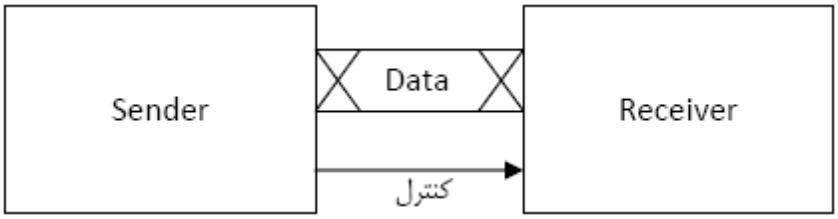
۱. سنکرون (همگام)



و اما چطور بفهمیم و مطمئن شویم که داده منتقل می‌شود.

در روش سنکرون به ازای هر clock اطلاعات انتقال میابد، این روش خوب نیست چون اولاً سیم clock می‌خواهیم و ثانیاً بعضی دستگاهها clock پذیر نیستند (مثل Mem)

۲. آسنکرون (ناهمگام)



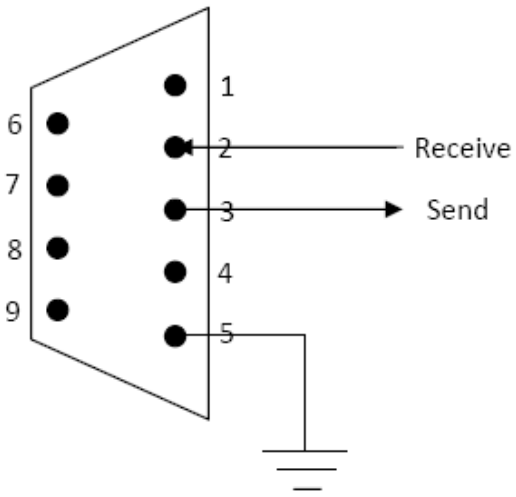
البته هر کدام از دستگاهها می‌توانند clock محلی داشته باشند.

انتقال اطلاعات به دو صورت ممکن است :

- سری(serial): همیشه یک بیت انتقال می‌یابد.
- موازی(parallel): چند بیت همزمان انتقال می‌یابد.

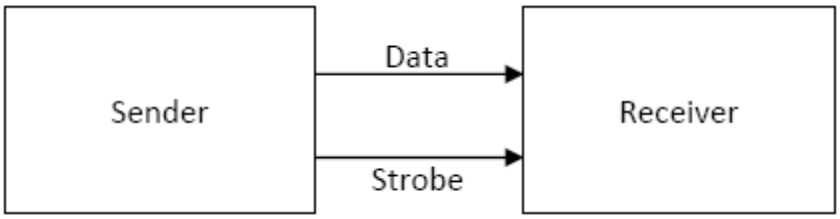
هر نوع کانکشنی پروتوکل مخصوص به خود را دارد.

ارتباط سری: (RS 232)



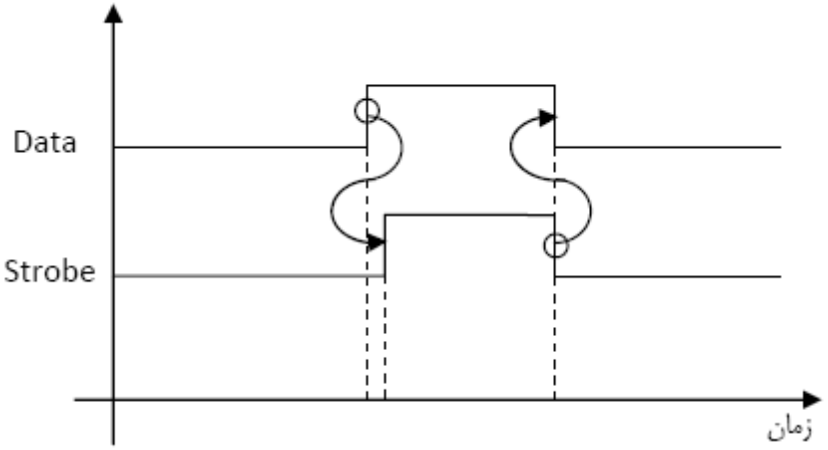
برای ارتباط باید بیت شماره ۳ sender را به بیت ۲ receiver وصل کنیم و زمین‌ها را باید به هم وصل کرد. اگر قرار بود ارتباط دو طرفه باشد بیت ۳ گیرنده را نیز به بیت ۲ فرستنده وصل می‌کنیم. به کابلِ چنین اتصالی cross می‌گویند که به دلیل عبور دو سیم از روی هم است. اگر اتصال به صورت cross نباشد (یعنی بیت ۲ و ۳ یک دستگاه به بیت ۲ و۳ دستگاه دیگر وصل شود) اتصال به منزله رابط است.

اکنون قصد داریم در مورد اتصالات از دیدگاه آسنکرون و سنکرون صحبت کنیم ولی کافیست آسنکرون سری را مورد بررسی قرار دهیم چون حالت موازی را میتوان با افزایش پهنای بیت حالت سری شبیه سازی کرد.



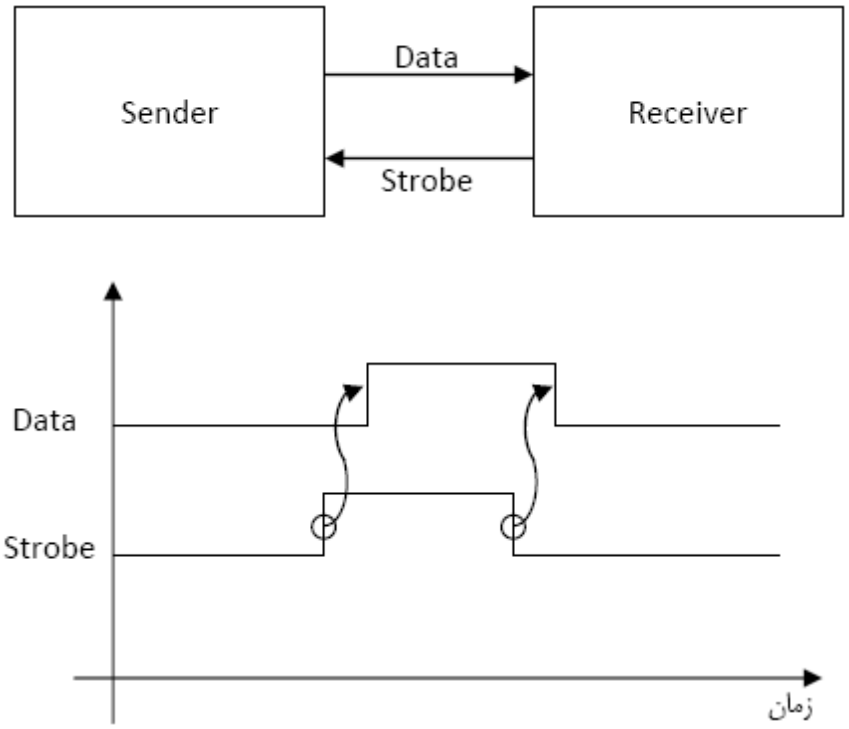
الف) ارتباط آسنکرون با استفاده از سیگنال strobe

- به ابتکار فرستنده
- به ابتکار گیرنده



خط strobe برای اینست که فرستنده به گیرنده خبر بدهد که مقدار داده معتبر است.

به چنین ارتباطی آسنکرون strobeدار به ابتکار فرستنده می‌گویند. (یا به عبارتی فرستنده initiator است). حالت دیگری نیز وجود دارد که گیرنده مشخص می‌کند که چه زمانی داده را معتبر فرض می‌کند که به آن strobeدار به ابتکار گیرنده می‌گویند. یعنی :



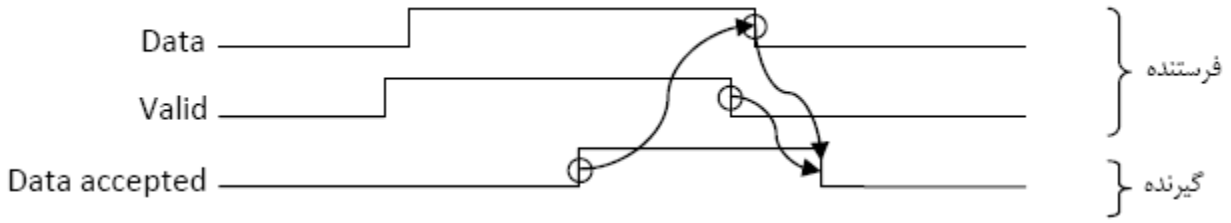
در این روش گیرنده اعلام می‌کند که آماده است تا داده را بگیرد. گیرنده داده‌ای را می‌گیرد که در زمان ۱ بودن strobe داده پایدارتر بوده. دقت کنیم که ارتباط، سری و یک بیتی است و در هر مرحله فقط یک بیت منتقل میشود پس برای انتقال هر بیت باید strobe یکبار کار کند. مزیت این روش اینست که فقط به دو سیم برای ارتباط نیاز دارد و عیب آن اینست که فرستنده هیچ کاری به این ندارد که گیرنده داده را استفاده کرده یا خیر. لذا روش دیگری نیز پدید آمد:

ب) با استفاده از تکنیک Handshaking (دست دهی)

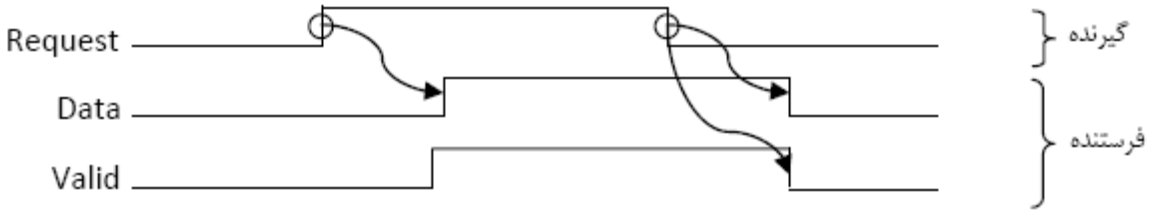
- به ابتکار فرستنده
- به ابتکار گیرنده

در این حالت سیگنال data valid نیز داریم.

درحالت به ابتکار فرستنده:



در حالت به ابتکار گیرنده:



در این روش داده وقتی برداشته می‌شود که توسط گیرنده دریافت شده باشد. در روش strobeدار اگر گیرنده busy باشد داده را نمی‌گیرد و آن را از دست می‌دهد.

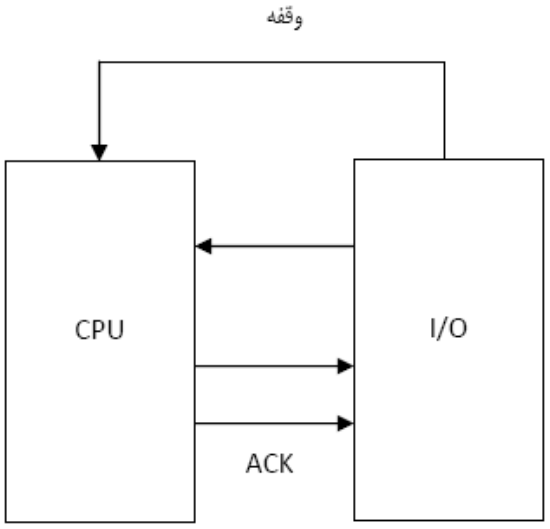
وقفه و I/O:

دستگاه‌های لَخت کار CPU را سخت می‌کنند چون CPU باید زمان زیادی را منتظر انجام درخواستش از آنها بماند لذا باید دنبال روشی بود که لازم نباشد پردازنده منتظر بماند بلکه این دستگاه‌های I/O باشند که وقتی با CPU کار داشتند یا کار درخواستی آن را انجام دادند به CPU اطلاع دهند تا رسیدگی کند.

برای برقراری ارتباط بین CPU و I/O دو روش وجود دارد:

۳. Polling (سرکشی): یک flag باشد که هر وقت توسط دستگاه I/O، Set شد (یعنی ۱ شد) CPUبه سراغ آن برود که لازمه اش اینست که CPU در یک حلقه بی نهایت، سیگنالِ وقفه را چک کند که عملاً کارایی را از دست می‌دهد.

۴. وقفه: یعنی هر وقت I/O با CPU کار داشت یک وقفه به آن می‌دهد و داده را به CPU می‌فرستد سپس CPU سیگنال ACK را که به معنی پذیرش درخواست است به I/O ارسال می‌کند و همزمان روتین وقفه اجرا می‌شود که داده را Handle کند.



دو نوع وقفه داریم :

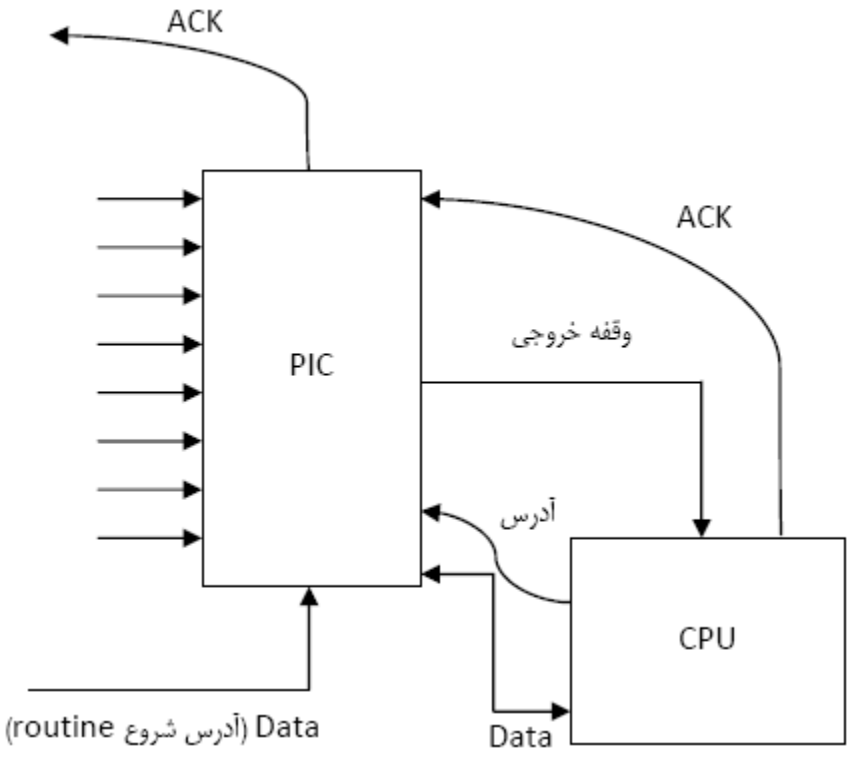
- سخت افزاری(External)
- نرم افزاری (Internal)

منظور از وقفه‌های سخت افزاری آنهایی است که برایشان سیم کشی مستقیم سخت افزاری انجام شده و اولویت بیشتری نسبت به وقفه‌های نرم افزاری دارند. وقفه‌های نرم افزاری دو دسته اند :

- ✓ وقفه‌هایی که در خود CPU رخ می‌دهند مثل تقسیم بر صفر.
- ✓ وقفه‌هایی که در زبان اسمبلی با دستور INT توسط خود کاربر ایجاد می‌شوند.

وقفه‌های نوع دوم مثل اجرای یک subroutine عمل می‌کنند با این تفاوت که نسبت به subroutineهای معمولی اولویت و الزام بیشتری برای اجرا دارند. به طور مثال بهتر است دستور نوشتن زمان روی مانیتور توسط روتین وقفه اجرا شود.

سوال اینجاست که برای وقفه‌های سخت افزاری چند پایه در CPU در نظر بگیریم؟ بدیهی است که هر چه تعداد دستگاه‌های I/O بیشتر باشد پایه‌های بیشتری لازم است و مدارها پیچیده تر می‌شوند. لذا از یک پایه وقفه استفاده شد که توسط قطعه‌ای به نام PIC(Programmable Interrupt Control) قابل انشعاب به چند وقفه است. به عبارت دیگر PIC چندین پایه وقفه سخت افزاری را به یکی تبدیل می‌کند. دیاگرام آن چنین است:

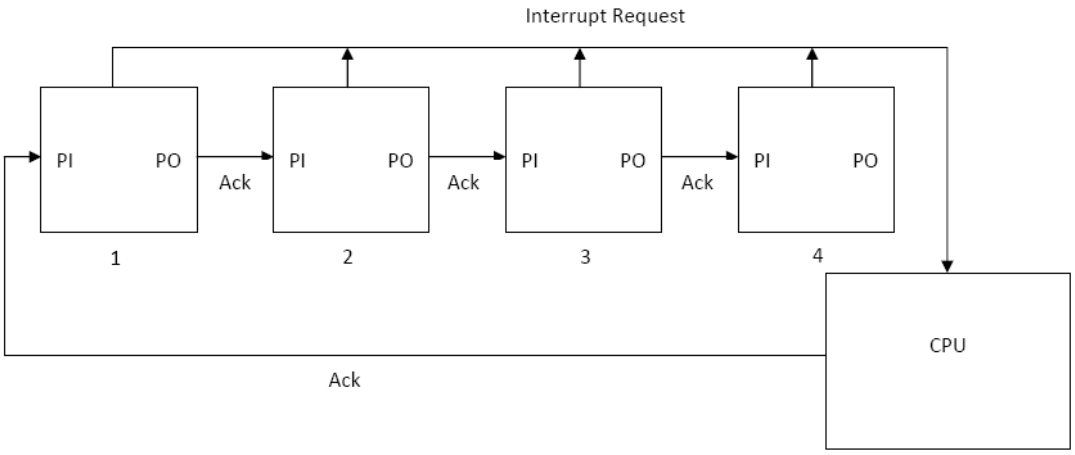


Handle کردن (راه اندازی) وقفه به دو صورت است:

- ✓ Vectored (برداری): قابلیت آدرس دهی به وقفه‌ها را دارد و هر وقت وقفه آمد آدرس آن در کلاک بعدی می‌آید.
- ✓ Non Vectored (غیربرداری): هر وقت وقفه آمد آدرس آن به صورت اتوماتیک از یک محل ثابت خوانده می‌شود.

اگر برخی پایه‌های PIC همزمان ۱ شوند، اولویت بندی می‌کنیم و معمولاً بالایی‌ها اولویت بیشتری دارند که در نتیجه عملاً PIC مثل یک I/O عمل می‌کند.

این روش برای زمانی است که می‌خواهیم وقفه‌ها به صورت موازی از PIC به CPU بروند. می‌شود به صورت سری هم وقفه‌ها را منتقل کرد که به آن Daisy Chain می‌گویند.



در این حالت درخواست به CPU می‌رود سپس CPU، سیگنال ACK را ارسال می‌کند. این سیگنال به اولین I/O میرسد اگر این دستگاه همانی باشد که وقفه را ارسال کرده ACK را می‌گیرد و ادامه روند وقفه را انجام می‌دهد در غیر اینصورت سیگنال ACK را به دستگاه بعدی در زنجیره I/Oها می‌فرستد و آن دستگاه نیز همین روند را طی می‌کند تا بالاخره سیگنال به دستگاه موردنظر برسد. البته در این روش فقط به یک پایه وقفه نیاز داریم.

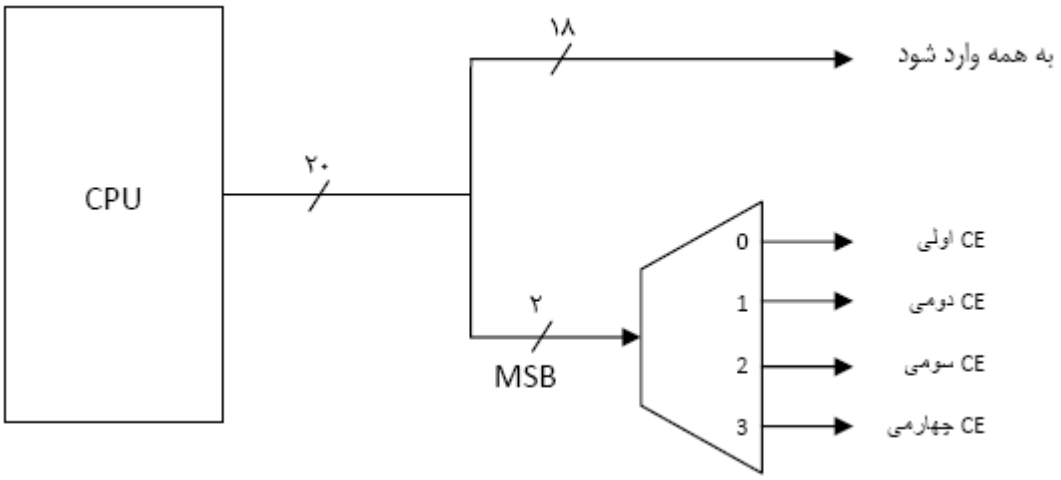
به وسیله علائم اختصاری زیر کامپیوترها به دسته‌های زیر تقسیم می‌شوند :

D:data I: instruction M: multiple S: single

- ✓ SISD: یعنی با یک دستور روی یک داده کار کنیم. (کامپیوتر پایه)
- ✓ SIMD: یعنی با یک دستور چندین داده دستکاری می‌شوند. مثل ضرب ماتریس‌ها.
- ✓ MISD: چند دستور ولی با یک داده مثل محاسبه مساحت دایره.
- ✓ MIMD: یعنی همزمان چند دستور با چند داده کار می‌کند. (البته هنوز چنین کامپیوتری ساخته نشده)

یک مسئله: فرض کنید CPU گذرگاه آدرس ۲۵تایی دارد اما حافظه ۱۸ بیت ورودی دارد. چه کار کنیم؟

جواب: کافیست ۴ تا از این حافظه‌ها را کنار هم بگذاریم و از دو بیت در فرمت آدرس، برای انتخاب بین این ۴ تا در هنگام کار استفاده کنیم. بدین صورت :



ضمیمه ۱:

آشنایی با نرم افزار

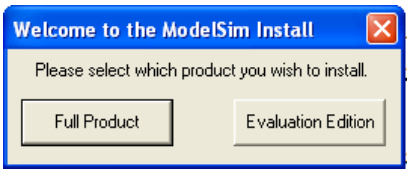
ModelSim, SimWattch

در این ضمیمه نحوه نصب، اجرا و کار با برنامه های ModelSim و SimWattch شرح داده شده است

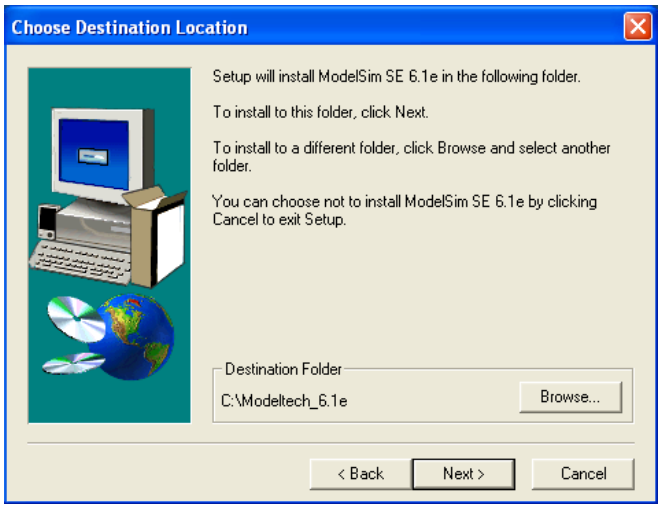
ModelSim

طریقه نصب و راه اندازی نرم افزار ModelSim

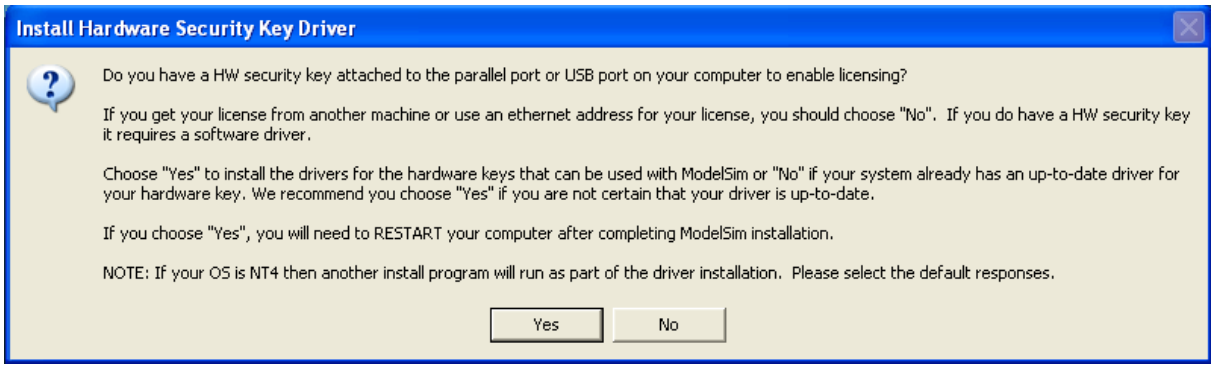
نرم‌افزار ModelSim یک شبیه‌ساز سخت‌افزار است. این شبیه‌ساز هر سه زبان سخت‌افزار verilog، VHDL و SystemC را پشتیبانی می‌کند. برنامه‌های توصیفی نوشته شده بازگردانی شده و شبیه‌سازی می‌شوند. نسخه‌ای از این نرم‌افزار که ما مورد استفاده قرار می‌دهیم ModelSim SE 6. 1 است. برای نصب این نرم‌افزار ابتدا نصب‌کننده را اجرا می‌کنیم. برای نصب دو گزینه Full و Evaluation وجود دارد. برای نصب در حالت Full به فایل License نیاز داریم، درحالی که نسخه Evaluation رایگان است. گزینه Full Product را انتخاب می‌کنیم.



بعد از قبول مراتب کپی رایت به پنجره انتخاب محل نصب می‌رسیم. این شاخه از قبل به صورت C:\modeltech_6. 1e است.

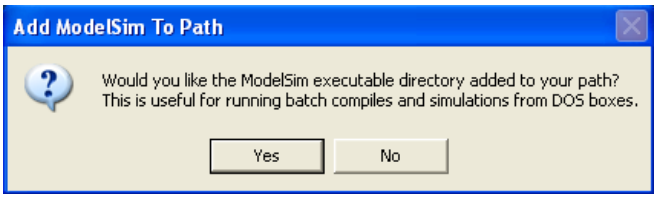


بعد از طی این مراحل، نصب نرم‌افزار آغاز می‌شود. بعد از پایان کپی کردن فایل‌ها، پنجره زیر با عنوان Install Hardware Security Driver ظاهر می‌شود.



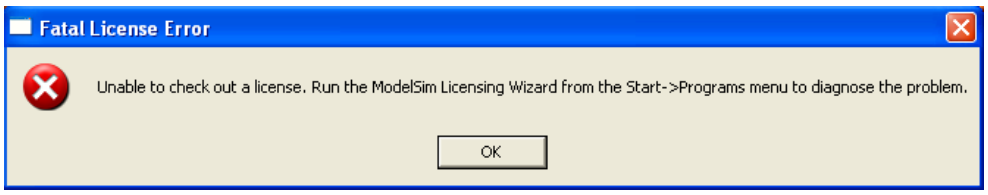
این مرحله برای فعال سازی Licensing است. از آن جا که ما فایل License خود را از Key Generator می‌گیریم، این مرحله لازم نیست و می‌توانیم گزینه No را انتخاب کنیم.

در مرحله بعد پنجره زیر ظاهر می‌شود. گزینه Yes را انتخاب می‌کنیم. این کار باعث می‌شود که بتوانیم در محیط command-lineاز دستورات ModelSim استفاده کنیم. در واقع مسیر "C:\modeltech_6. 1e\win32" به متغیر PATH اضافه می‌شود.



اگر بخواهیم به صورت دستی این متغیر را تغییر دهیم باید وارد System Properties شده و متغیر مربوطه را ویرایش کنیم.

حالا اگر ModelSim را اجرا کنیم با اعلان خطای زیر مواجه می‌شویم :

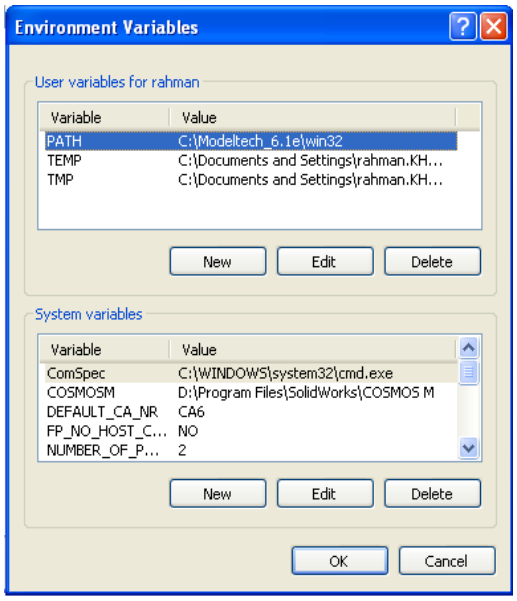


این اعلان خطا به ما می‌گوید که هنوز فایل License را ایجاد نکرده ایم. بنابراین به صورت زیر عمل می‌کنیم:

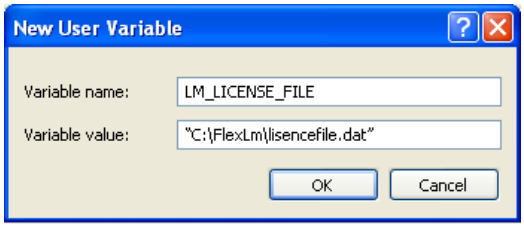
برای ایجاد فایل License از برنامه MentorKG. exe استفاده می‌کنیم. این برنامه یک ورودی Lisence. src می‌گیرد و خروجی Lisencefile. dat را به ما می‌دهد. دستور زیر این کار را انجام می‌دهد:

MentorKG. exe -i license. src -o licensefile. dat –hd

یک پوشه با نام FlexLm در درایو C ایجاد کرده و فایل Lisencefile. dat را در آن پوشه کپی می‌کنیم. سپس یک متغیر محیطی با نام LM_LISENCE_FILE ایجاد کرده و مقدار آن را برابر با "C:\FlexLm\lisencefile. dat" قرار می‌دهیم. برای این کار وارد System Properties شده و از منوی Advanced گزینه Environment variable را انتخاب می‌کنیم. پنجره مقابل ظاهر می‌شود:



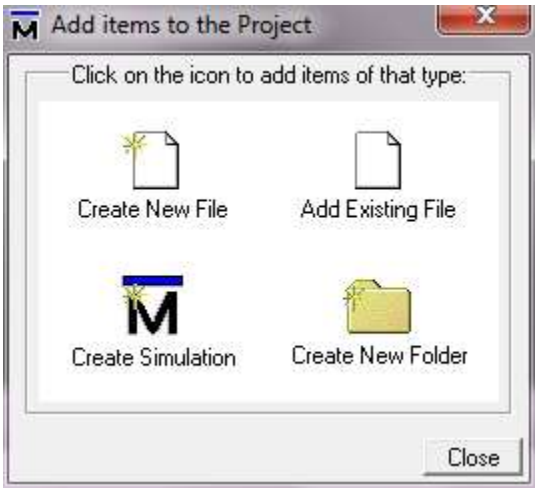
حالا در قسمت User یک متغیر جدید با نام LM_LICENSE_FILE می‌سازیم و مقدار آن را برابر با "C:\FlexLm\lisencefile. dat" قرار می‌دهیم.



بعد از تأیید پنجره متغیرهای محیطی می‌توانیم ModelSim را اجرا کنیم.

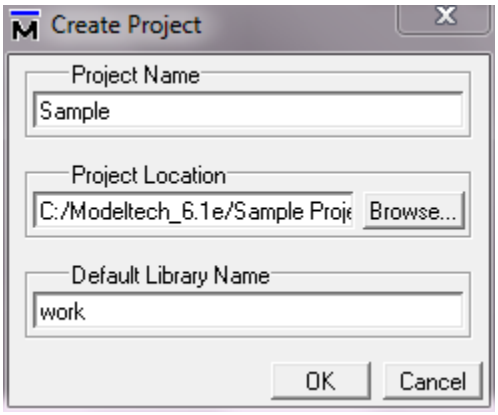
آشنایی با محیط نرم‌افزار و شیوه کار با آن

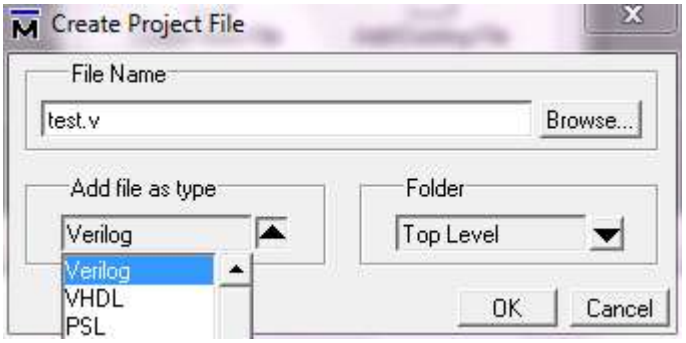
هنگام باز کردن نرم افزار Model Sim با پنجره‌ی زیر مواجه مـی‌شـوید. بـرای ایجـاد یـک فایـل verilog بایـد گزینه‌ی Create New File را انتخاب کنید.



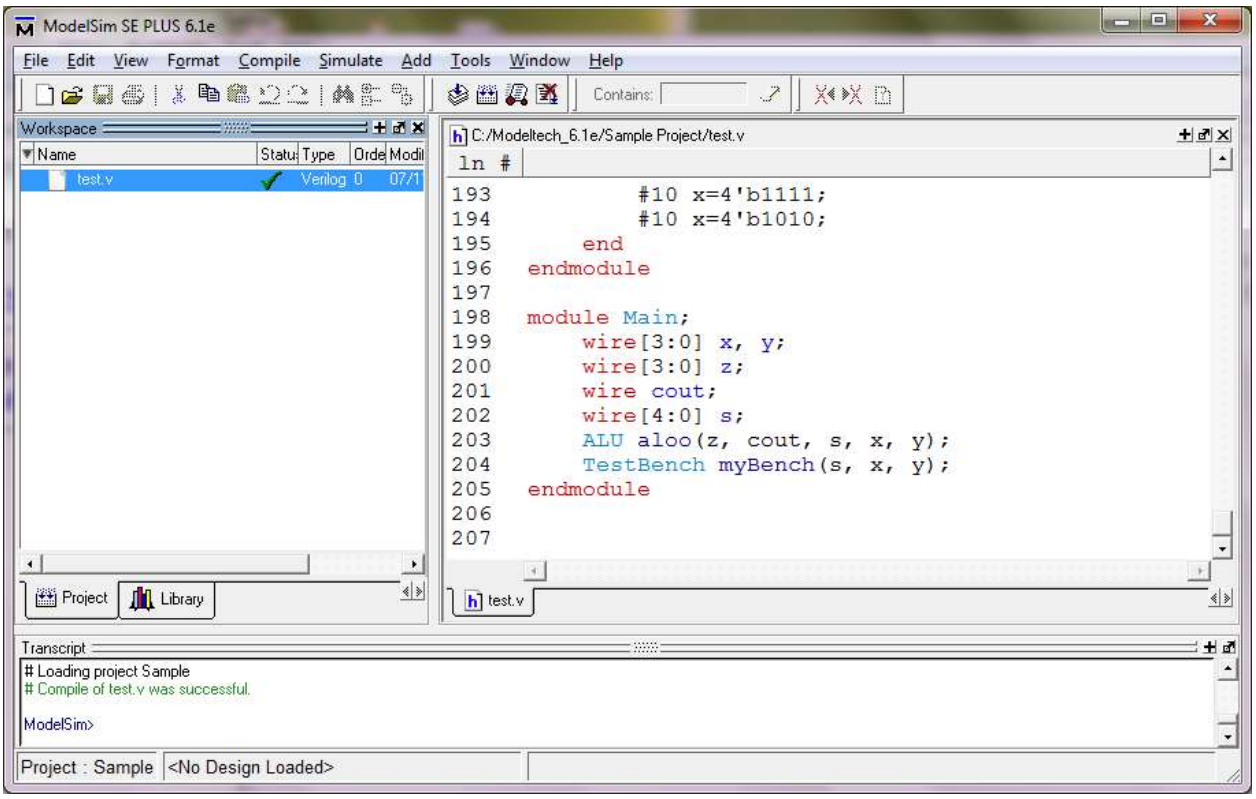
قبل از ساختن فایل باید یک project تعریف کنید. نرم افزار به طور اتوماتیک پنجره‌ی زیر را می‌آورد. هم چنین می‌توانید در منوهای بالا بر Project <- New <- File کلیک کنید.

در هنگام تعریف project علاوه بر تعیین نام، باید مکان ذخیره سازی فایل‌ها را نیز مشخص کنید. سـپس یـک فایل verilog بسازید. دقت کنید که پسوند را در نام فایل ذکر کنید و نوع فایل را از حالت عـادی (VHDL) بـه حالت مورد نظر (verilog) تغییر دهید.

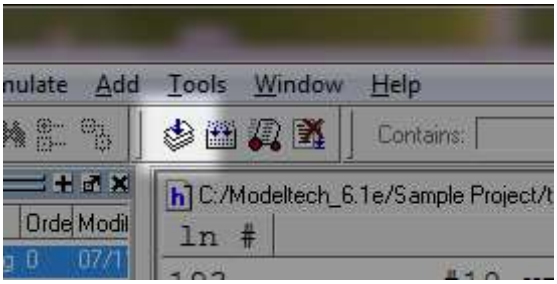




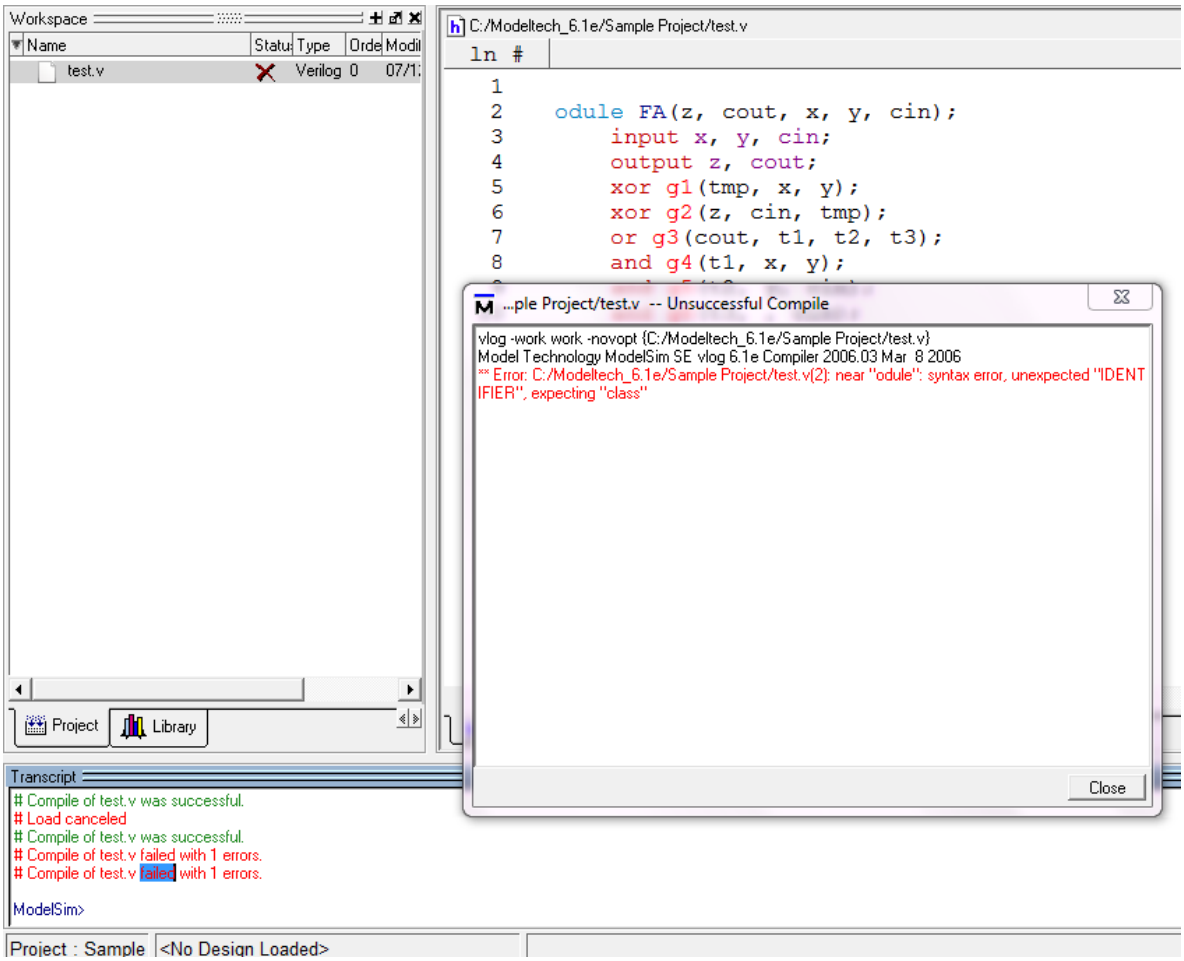
پس از ساختن فایل verilog با محیط برنامه نویسی زیر رو به رو می‌شوید:



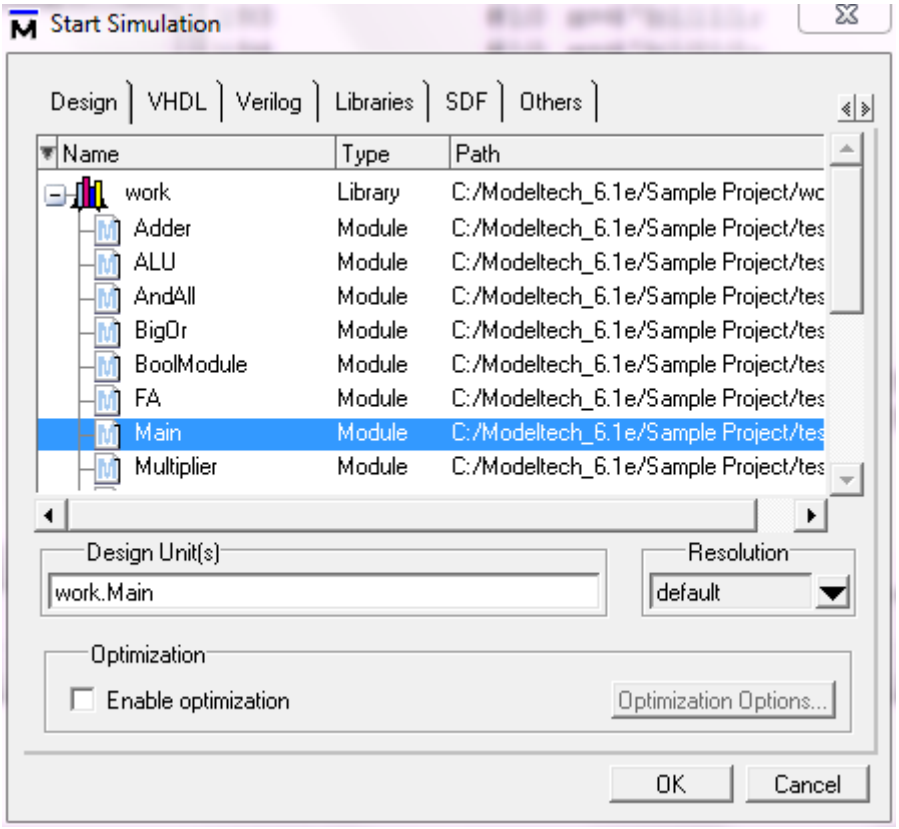
در پنجره‌ی سمت چپ لیست فایل‌های پروژه آمده است. می‌توانید از طریق منوها فایل‌ها بیشتری نیز اضافه کنید.

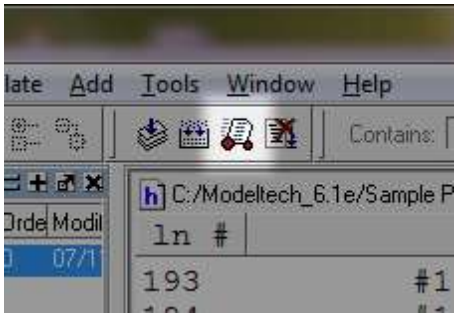


پس از آنکه برنامه شما کامل شده است باید آن را compile کنید. این کار را می‌توانید از طریق گزینه‌ی Compile در منوها انجام دهید، یا دکمه‌ی مخصوص آن را فشار دهید. (عکس مقابل را ببینید). در صورتی که برنامه عاری از مشکلات زبانی باشد، در پنجره‌ی Transcript پیغام سبز موفقیت compile می‌دهد. در غیر این صورت پیغام قرمز می‌دهد و تعداد خطاها را اعلام می‌کند. برای مشاهده‌ی خطاها کافی است بر روی پیغام double-click کنید. پنجره‌ای باز می‌شود که برای هر خطا یک پیغام قرمز در آن داده شده. با کلیک کردن بر روی هر یک از پیغام‌ها می‌تونید، محل بروز خطا را در برنامه تان مشاهده نمایید.



در صورتی که compile با موفقیت انجام شد و قصد شبیه سازی یا اجرای برنامه خود را دارید، از طریق Simulate -> Start Simulation در منوها، یا از طریق دکمه‌ی مخصوص شبیه سازی (عکس مقابل را، می‌توانید وارد محوطه‌ی شبیه سازی می‌شوید. قبل از اجرا، باید تابع اجرایی را مشخص کنید. مثلا تابع Main را انتخاب می‌کنید و OK را می‌زنید.

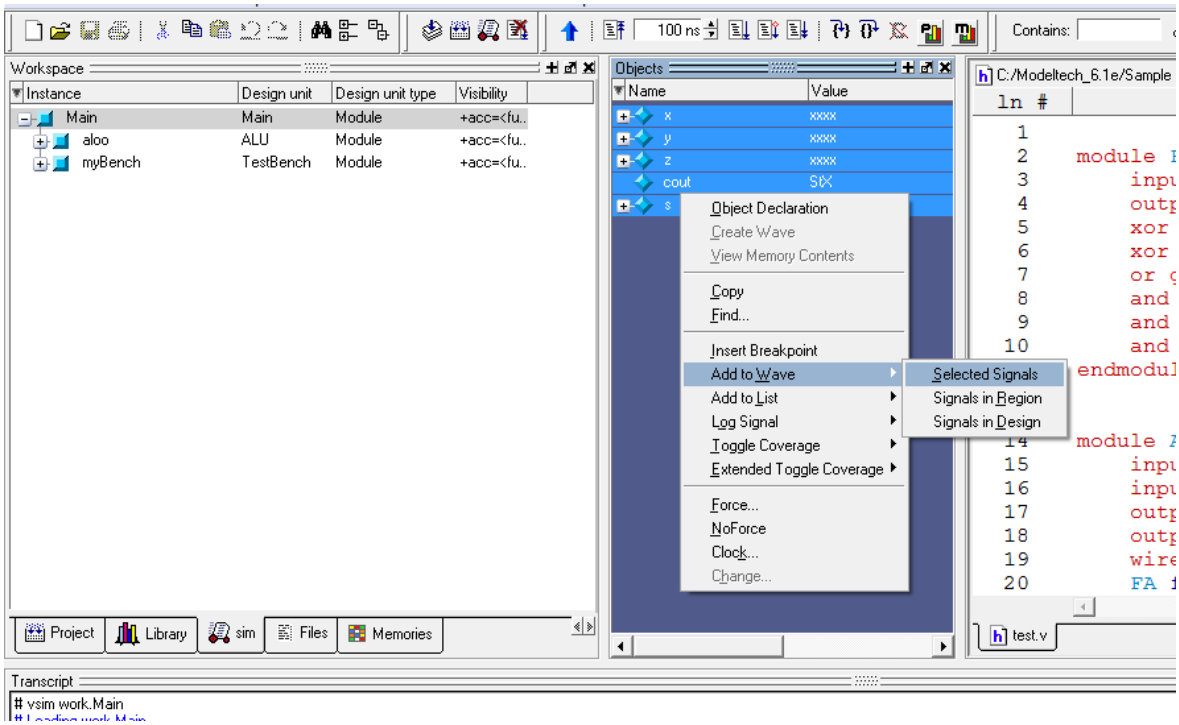




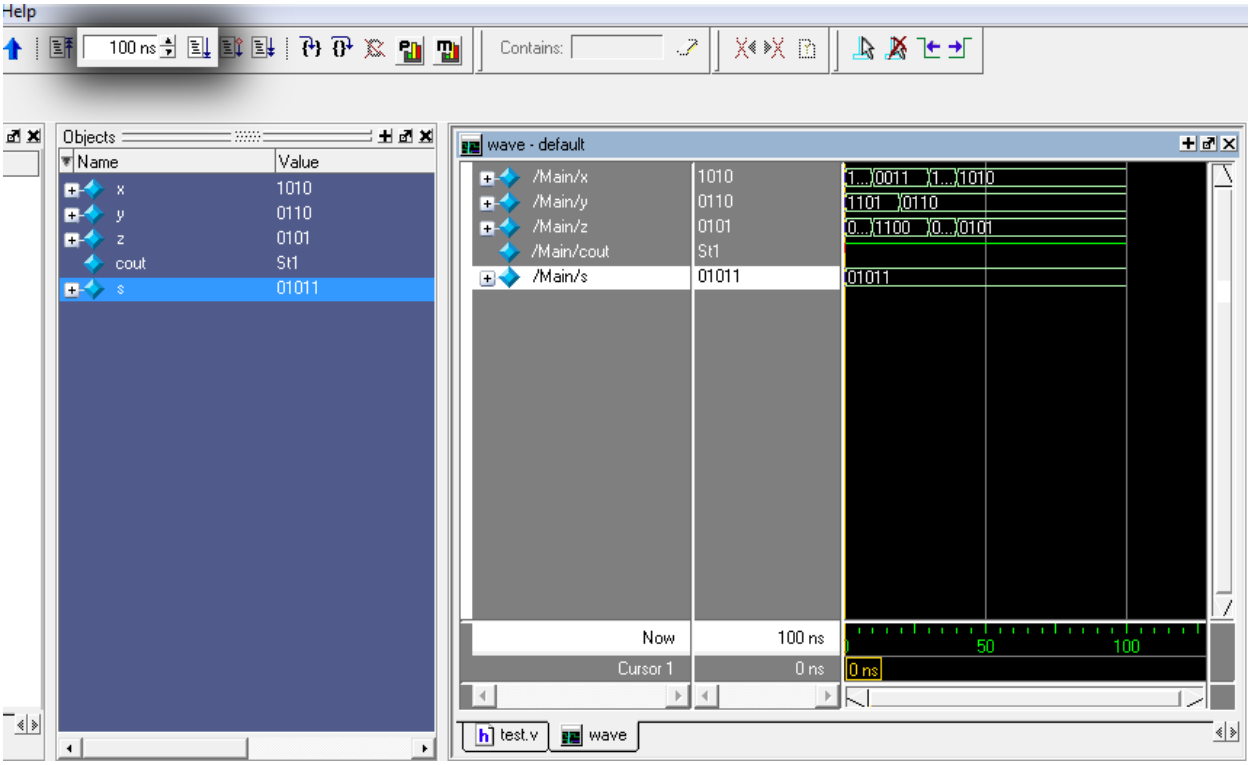
از این شبیه ساز می‌توانید برای debug کردن برنامه‌های توصیفی خود استفاده کنید. قسمت‌های ترتیبی به صورت زمانی اجرا می‌شوند و می‌توان خط به خط آن‌ها را اجرا کرد. روی این خط‌ها می‌توانید نقاط استراحت(breakpoint) تعریف کنید و یا برای متغیرها نقاط استراحت تعیین کنید. این به معنی آن است که هر وقت متغیر مورد نظر تغییر کرد، برنامه متوقف می‌شود و شما می‌توانید مقادیر جدید را مشاهده کنید.

شبیه ساز ModelSim ابزارهایی برای دنبال کردن برنامه حین اجرا دارد که در هنگام Debug کردن بسیار مفید هستند. از آن جمله می‌توان محیط‌های wave و List و Dataflow را نام برد. هرکدام از این ابزارها کارایی ویژه خود را دارند. اما ابزار wave از آن جا که می‌تواند مقدار متغیرها را در مقاطع مختلف زمانی نشان دهد، بیشتر مورد توجه است. روش کار با این پنجره به این صورت است که ابتدا باید از view -> Debug Windowsگزینه wave را انتخاب کنید. پنجره زیر ظاهر می‌شود. حالا برای نمایش متغیرها، در پنجره objects روی متغیرها کلیک راست نموده و به صورت زیر آن‌ها را به صفحه موج اضافه کنید. البته این کار را با Drag and Drop نیز می‌توان انجام داد.

پنجره موج به صورت زیر است و در آن قسمتی مربوط به متغیرها و قسمتی مربوط به مقادیر و قسمتی مربوط به موج است. در قسمت موج یک خط‌کش هست که نشان دهنده زمان است. مثلا در شکل زیر در زمان ۱۰۰



نانوثانیه هستیم. این زمان البته در حین debug در نوار وسیله شبیه‌ساز نیز درج می‌شود. می‌توانید مقادیر را به صور مختلف دودویی یا در مبناهای دیگر نمایش دهید. مقادیر نامشخص(x) و امپدانس بالا(HiZ) با رنگ‌های دیگر نمایش می‌شوند.



می‌توانیم برای خواناتر شدن پنجره موج از نوارهای تقسیم کننده استفاده کنیم که هر قسمت مربوط به یک بخش را جدا کنند. این کار را می‌توان با کلیک راست کردن بر روی قسمت متغیرهای پنجره موج و انتخاب گزینه Insert Divider انجام داد.

Sim-Wattch

معرفی برنامه sim-wattch :

برنامه sim-wattch شبیه سازی جهت ارزیابی و ارائه جزئیات مربوط به کارایی سخت افزار و پردازنده طراحی شده با سیستم شامل حافظه دو سطحی(L1-L2) می‌باشد. همچنین این برنامه توانایی یافتن تاخیر موجود در همه عملیات خط لوله (Pipeline) را نیز دارد.

طریقه نصب برنامه sim-wattch :

این برنامه نیازی به نصب ندارد ؛ فقط کافی است که پوشه فشرده شده را از حالت فشردگی خارج کنیم. فایل‌های مورد نیاز جهت انجام عملیات مورد نظر در پوشه موجود هستند. قابل ذکر است که در نسخه مربوط به ویندوز برخلاف نسخه تحت لینوکس, مشکلاتی اعم از نصب, اجرا بدلیل سادگی کار و آشنایی بیشتر با ویندوز کمتر مشاهده می‌شود.

نحوه کارکردن با شبیه ساز :

در ابتدا جهت شبیه سازی پردازنده مورد نظر باید ورودی‌هایی به برنامه sim-wattch داده شود. به عبارت دیگر نتایج خروجی پس از تحلیل بر روی برنامه‌ی کاری داده شده بدست می‌آید. برنامه‌های کاری مختلفی بعنوان استاندارد کاری جهت انجام تحلیل‌های مربوط به این شبیه ساز وجود دارند که از جمله مهمترین آنها می‌توان به gcc,vpr,mcf,art اشاره کرد. همچنین لازم به ذکر است که کلیه اطلاعات مورد نیاز ما جهت تحلیل خروجی‌های مربوط به حافظه نهان در فایل خروجی موجود است. اطلاعاتی از جمله تعداد کل دسترسی‌ها (access), تعداد برخوردها (hit), اعداد و درصد فقدان ها(miss) ,تعداد و درصد جایگزینی‌ها (Replacement) , میزان مصرف برق و غیره از جمله اطلاعاتی است که به همراه حافظه نهانی که جهت تحلیل‌های ما مورد نیاز می‌باشد(که در فایل‌های خروجی تحت نام های dl, il و ul آورده شده اند) در فایل خروجی وجود دارند.

حال باید ببینیم که ورودی باید به چه صورت باشد. برای دادن ورودی باید فایل sim-outorder. exe را اجرا کرده و با توجه به فرمان‌ها و آرگومان‌های موجود, ورودی موردنظر خود را تنظیم کنیم. نکته مهم در اجرای این فایل این است که باید در محیط cmd (Command Prompt) در ویندوز یا معادل آن در سیستم عامل لینوکس (که همان کنسول bash می‌باشد) فایل مذکور اجرا شود و پارامترهای موردنظر تنظیم شوند.

یکی از پارامترهای مهم جهت توصیف پردازنده, مشخصات مربوط به ظرفیت حافظه نهان, اندازه بلوک ها, cache line ها, مقدار k (که مربوط به k-way set associative cache است) می‌باشد. فرمت مشخص نمودن این مقادیر در مقابل حافظه نهان ul2 بصورت مقابل است :

ul2: cache line: block size: K: l

که cache line طبق رابطه مقابل بدست می‌آید :

Cache line = cache size /K * block size

حال اگر هرکدام از پارامترهای بالا متغیر باشد, طبیعتا متغیرهای وابسته آن در فرمت بالا نیز تغییر خواهند کرد.

یکی دیگر از پارامترهای ورودی نام فایل خروجی می‌باشد که باید همان ابتدا مشخص شود. در واقع این نام باید بصورت نسبی بر پایه فولدر جاری مشخص شود.

در قسمت زیر نمونه‌ای از ورودی را مشاهده می‌کنید. ورودی‌ها بعنوان آرگومان اجرایی فایل sim-outorder. exe داده می‌شوند. حافظه نهان موردنظر نیز ul2 می‌باشد. به آدرس فایل خروجی که با رنگی مجزا مشخص شده است, توجه بفرمایید :

توجه شود که اگر تعداد تست هایمان زیاد باشد, می‌توانیم فایل اجرایی با فرمت bat. (که فرمت batch file می‌باشد) ساخته و کلیه این ورودی‌ها را در آن کپی نماییم. سپس با قرار دادن این فایل در فولدر sim-wattch که فایل sim-outorder. exe وجود دارد و اجرای خودکار آن می‌توان بدون اتلاف وقت به خروجی‌ها دست یافت.

از جمله مهمترین نکاتی که باید مورد توجه قرار بگیرد این است که برنامه اجازه گرفتن بلوک‌هایی با اندازه کمتر از ۳۲ بایت را نمی‌دهد و خطایی بصورت ERROR: division by zero می‌دهد. پس لازم است که اندازه بلوک‌ها را از ۳۲ شروع نماییم.

پس از اجرای برنامه با ورودی داده شده, فایل خروجی در آدرس داده شده ساخته می‌شود. پس از مراجعه به فایل و باز نمودن آن (ترجیحا از ادیتورهایی پیشرفته مثل Em-editor و یا notepad++ استفاده شود تا هر خط بصورت مجزا نشان داده شود.) کلیه اطلاعات موجود می‌باشد.

در زیر نمونه‌ای از فایل خروجی پس از جستجوی عبارت ul2 آورده شده است. همانطور که مشاهده می‌شود, اطلاعات موردنظر روبروی هر گزینه آورده شده است :

```
dl1.repl_rate                0.0275 # replacement rate (i.e., repls/ref)
dl1.wb_rate                  0.0057 # writeback rate (i.e., wrbks/ref)
dl1.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
ul2.accesses               2638849 # total number of accesses
ul2.hits                   2587260 # total number of hits
ul2.misses                   51589 # total number of misses
ul2.replacements             21816 # total number of replacements
ul2.writebacks                8661 # total number of writebacks
ul2.invalidations                0 # total number of invalidations
ul2.miss_rate                0.0195 # miss rate (i.e., misses/ref)
ul2.repl_rate                0.0083 # replacement rate (i.e., repls/ref)
ul2.wb_rate                  0.0033 # writeback rate (i.e., wrbks/ref)
ul2.inv_rate                 0.0000 # invalidation rate (i.e., invs/ref)
itlb.accesses              59061458 # total number of accesses
itlb.hits                  59046097 # total number of hits
itlb.misses                   15361 # total number of misses
```

پس از دریافت داده‌های مربوطه و تجزیه و تحلیل داده‌ها و رسم نمودار می‌توان مقادیر مربوط به ورودی‌های مختلف را با یکدیگر مقایسه نمود.

ضمیمه ۲:

سوالات نمونه

در این بخش تلاش شده تا مسائلی با پاسخ آنها جمع آوری شود.

اعداد اعشاری:

۱. اعداد زیر را با توجه به **استاندارد IEEE754** به صورت ممیز شناور بنویسید. (منبع: Patterson,COD,Page195)

a. 0. 75- (3/4-) با دقت ساده(Single Precision)

b. 0. 75- (3/4-) با دقت مضاعف(Double Precision)

c. عدد زیر که در مبنای دودویی و به صورت ممیز شناور و بادقت ساده می‌باشد را به مبنای دهدهی تبدیل نمایید.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

۲. با استفاده از الگوریتم‌های موجود در اعدادهای اعشاری دستورات محاسباتی زیر را با استفاده از ممیز شناور انجام دهید. (Patterson, COD)

اعداد: 0. 5 و 0,4375- که در مبنای ۱۰ می‌باشند.

a) عملگر جمع

b) عملگر ضرب

۳. فرمت ممیز شناوری را در نظر بگیرید که ۸ بیت برای توان و ۲۳ بیت برای قسمت شناور می‌باشد حال اعداد زیر به این فرمت تبدیل نمایید. (Stalling,COD,Page 327)

a. 720-

b. 0. 645

۴. دلایل استفاده از بایاس را بنویسید. (Patterson, COD ,Page 194)

a. اگر توان عددی در پایه‌ی ۲ باشد(B=2) و با ۶ بیت نمایش داده شود. مقدار بایاس چند می‌باشد؟(Stalling,COD,Page 327)

b. اگر توان عددی در پایه‌ی ۸ باشد(B=8) و با ۷ بیت نمایش داده شود. مقدار بایاس چند می‌باشد؟(Stalling,COD,Page 327)

۵. اعداد زیر با استفاده از استاندارد IBM 32bit , که در پایه ۱۶ می‌باشد و ۷ بیت را برای توان در نظر می‌گیرد را به اعداد ممیز شناور تبیل نمایید. (Stalling,COD,Page 327)

a. ۰. ۱

b. ۰. ۵

c. ۱/۶۴
d. ۰ .۰
e. ۰- .۱۵
f. ۴ .۵ × ۱۰ ^-۷۹
g. ۲ .۷ × ۱۰ ^۷۵
h. ۶۵۵۳۵

۶. با استفاده از الگوریتم تقسیم دو عدد اعشاری دو عدد زیر بر هم تقسیم نمایید. عدد اول: ۱. ۲۵
عدد دوم: ۴. ۷۵
(عدد اول بر روی عدد دوم می‌باشد و همچنین دقت دستگاه ۴ بیت می‌باشد)

۷. حال سرریز و ته ریز را تعریف نمایید و سپس بگویید در عملیات‌های مختلف چه نوع خطایی ممکن است رخ دهد؟

حل مسائل:

۱) در این سوال باید مراحل زیر را گام به گام طی نمود

a) در ایتدا عدد را به صورت باینری در می‌آوریم: -0. 75 = -0. 11

b) حال این عدد را هنجار می‌کنیم: $-0.11 = -1.1 \times 2^{-1}$

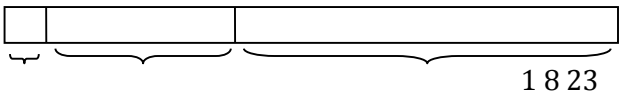
c) فرمت اعداد در ممیز شناور به شکل زیر می‌باشد

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - 127)}$$

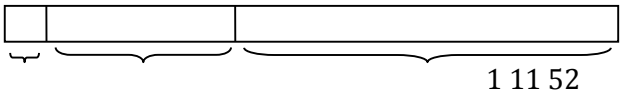
که در Exponent مقدار بایاس به آن اضافه شده است بنابراین در فرمول آن این مقدار از آن کم کرده است

d) حال با توجه به تقسیم بندی بیت‌ها در دو حالت مختلف می‌توانیم عدد خود به فرمت ممیز شناور در بیاوریم.

Single Precision (دقت ساده): 32 bits



Double Precision (دقت مضاعف):64 bits



a) در حالت دقت ساده اعداد هرقسمت برابر زیر می‌شود.

S = 1

Exponent = -1 + 127= 01111110

Fraction = 10000000000000000000000

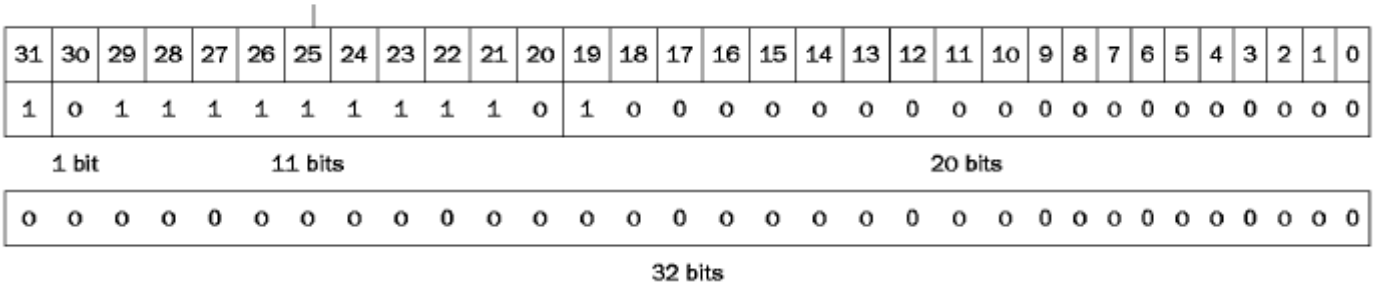
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1 bit				8 bits					23 bits																						

b) در حالت دقت مضاعف هر قسمت برابر زیر می‌شود.

S = 1

Exponent = 0111111110

Fraction = همانند شکل زیر



c) در این حالت جزییات عدد ما برابر است با

Exponent – Bias = 129 – 127 = 2

Fraction = 0. 01

Number = $-1.01 \times 2^2 = -1.25 \times 4 = -5.0$

۲) قبل از وارد شدن به الگوریتم‌های این قسمت باید اعداد را هنجار نماییم. و سپس با استفاده از الگوریتم‌های موجود در جزوه سوال را حل می‌نماییم.

$0.5 = (0.1)_2 = (1.0 \times 2^{-1})_2$

$-0.4375 = (-0.0111)_2 = (-1.11 \times 2^{-2})_2$

a) قدم اول: چک کردن صفر بودن اپرند ها

قدم دوم: مقایسه نماها: در این حالت عدد با نمای بزرگتر, عدد دوم می‌شود.

قدم سوم: شیفت دادن عدد با نمای کوچکتر به تعداد اختلاف نماها: به این ترتیب عدد دوم برابر است با $(-0.111 \times 2^{-1})_2$

قدم چهارم: اعشار را جمع می‌کنیم :

$(1.0 \times 2^{-1})_2 + (-0.111 \times 2^{-1})_2 = (-0.001 \times 2^{-1})_2$

قدم چهارم: عدد را هنجار کنیم :

$(-0.001 \times 2^{-1})_2 = (-1.000 \times 2^{-4})_2$

b) در این حالت ما عدد هنجار شده را بدون بایاس در نظر می‌گیریم.

قدم اول: جمع کردن نماها: -1 + -2 = -3

و یا موقع استفاده از بایاس می‌توان نوشت

$$(-1 + 127) + (-2 + 127) - 127 = (-3 + 127)$$

قدم دوم: عمل ضرب انجام شود.

$$(1.0)_2 \times (-1.11)_2 = (-1.11)_2$$

قدم سوم: هنجار کردن عدد.

$$(-1.11 \times 2^{-3})_2 = (-1.11 \times 2^{-3})_2$$

۳)

a)

- در ابتدا عدد را به دودویی تبدیل می‌کنیم. ۱۰۱۱۰۱۰۰۰۰-
- هنجار کردن عدد: ۱. ۰۱۱۰۱۰۰۰۰ × $۲^{۹}$
- فیلد توان: با توجه به ۸ بیت بودن آن بنابراین ۱۲۸ برای بایاس به آن اضافه می‌کنیم که می‌شود ۱۰۰۰۱۰۰۱
- فیلد اعشار و یا مانتیس برابر است با: ۰۱۱۰۱۰۰۰۰۰۰۰۰۰۰۰۰۰
- فیلد علامت نیز برابر با یک می‌باشد.

b) در این حالت نیز باید مراحل بالا طی کنیم.

- تبدیل عدد به باینری:. ۰. ۱۰۱۰۰۱
- بیت علامت برابر با صفر است
- قسمت توان برابر با: ۰۱۱۱۱۱۱۱
- قسمت مانتیس هم برابر است با:۰۱۰۰۱

۴)

a) دو دلیل عمده برای این کار می‌باشد.

- درست کردن سخت افزاری برای مقایسه در سیستم مکمل دو مشکل و پیچیده می‌باشد. که با این کار تمام اعداد مثبت می‌شوند و مقایسه آنها ساده تر می‌گردد. (**پترسون**)

- در تمامی مدارهای سخت‌افزاری که برای **انجام محاسبات ریاضی وجود دارد, چک می-کنند که عملوندها صفر نباشند,** مثلا ممکن است که در یک محاسبه, عملوند صفر تعریف نشده باشد که با این کار دیگر به مشکلی بر نمی‌خوریم. (**جزوه**(؟))

b) برای تعیین بایاس مبنا مربوط نمی‌باشد. بنابراین برابر می‌شود با ۳۲

c) برای تعیین بایاس مبنا مربوط نمی‌باشد. بنابراین برابر می‌شود با ۶۴

۵)

برای حل این سوال همانند زیر حل می‌کنیم.

a.	1.0	$=$	$+1/16 \times 16^{1}$	$=$ 0 100 0001 0001 0000 0000 0000 0000 0000
b.	0.5	$=$	$+8/16 \times 16^{0}$	$=$ 0 100 0000 1000 0000 0000 0000 0000 0000
c.	$1/64$	$=$	$+4/16 \times 16^{-1}$	$=$ 0 011 1111 0100 0000 0000 0000 0000 0000
d.	0.0	$=$	$+0 \times 16^{-64}$	$=$ 0 000 0000 0000 0000 0000 0000 0000 0000
e.	-15.0	$=$	$-15/16 \times 16^{1}$	$=$ 1 100 0001 1111 0000 0000 0000 0000 0000
f.	5.4×10^{-79}	$\approx$	$+1/16 \times 16^{-64}$	$=$ 0 000 0000 0000 0000 0000 0000 0000 0000
g.	7.2×10^{75}	$\approx$	1×16^{63}	$=$ 0 111 1111 1111 1111 1111 1111 1111 1111
h.	65535	$=$	$16^{4} - 1$	$=$ 0 100 0100 1111 1111 1111 1111 0000 0000

۶) طبق الگوریتم باید مراحل زیر را طی کنیم.

عدد اول: ۱. ۲۵ عدد دوم: ۴. ۷۵

- تبدیل اعداد به دودویی: عدد اول: ۱. ۰۱ عدد دوم: ۱۰۰. ۱۱
- هنجار کردن اعداد: عدد اول: ۱. ۰۱ عدد ئوم:۱. ۰۰۱۱ × $۲^{۲}$
  - عدم صفر بودن اپرندها
  - تفریق دو نما: ۲ –۱ = ۱–
  - عمل تقسیم انجام شود. ۱. ۰۰۱۱/ ۱. ۰۱ = ۱. ۰۰۰۱

۷)

a. در ابتدا یه تعریف می‌پردازیم.

- زیر ریز (underflow): قبل از ممیز فقط رقم صفر وجود داشته باشد.

مثال

$$\begin{array}{l} 1.0011 \times 2^{10} \\ -\ 1.0010 \times 2^{10} \\ \hline 0.0001 \times 2^{10} \end{array}$$

- سر ریز (overflow): قبل از ممیز عددی بزرگتر از ۱ وجود داشته باشد.

$$\begin{array}{l} 1.0011 \times 2^{10} \\ +\ 1.0010 \times 2^{10} \\ \hline 10.0001 \times 2^{10} \end{array}$$

b. در جمع: فقط سرریز اتفاق می‌افتد.

در تفریق: صرفا ته ریز اتفاق می‌افتد

در ضرب: فقط سرریز اتفاق می‌افتد.

در تقسیم صرفا ته ریز اتفاق می‌افتد.

بخش ورودی خروجی

استخراج شده از کتاب‌های پترسون و استالینگ

۱۰- مراحل بررسی کردن یک وقفه را در یک سیستم بیان کنید

- And کردن منطقی وقفه جدید با interrupt mask field، برای بررسی اولویت وقفه جدید.
- انتخاب کردن وقفه با اولویت بالاتر.
- ذخیره کردن interrupt mask field رجیستر وضعیت (status register)
- تغییر دادن رجیستر interrupt mask field برای از کار انداختن تمامی وقفه‌های با اولویت برابر و یا اولویت کمتر(در صورتی که وقفه جدید اولویت پایین تری داشته باشد، این رجیستر بدون تغییر خواهد ماند.)
- ذخیره کردن حالت پردازنده برای شروع وقفه جدید
- ست کردن بیت وقفه (interrupt enable) برای پذیرش وقفه‌های با اولویت بالاتر
- انجام عملیات مربوط به وقفه جدید
- بازگردانی پردازنده و interrupt mask field به حالت قبل.

۲- فرض کنیم برنامه‌ای داریم که در ۱۰۰ ثانیه اجرا می‌شود، که ۹۰ ثانیه آن مربوط به زمان پردازنده و مابقی آن مربوط به ورودی/خروجی است. اگر سرعت پردازنده به مدت پنج سال با سرعت ۵۰٪ ارتقا پیدا کند، و سرعت ورودی/خروجی بدون تغییر باقی بماند، سرعت اجرا در پایان پنج سال چقدر خواهد بود؟

می دانیم که زمان اجرا برابر است با زمان پردازنده و زمان ورودی/خروجی:

۱۰۰ = ۹۰ + ورودی/خروجی ->ورودی/خروجی = ۱۰

زمان پردازنده در طول این پنج سال به ترتیب برابر ۱۲، ۶۰، ۴۰، ۲۷، ۱۸، ۹۰ خواهد بود. در نتیجه در پایان پنج سال سرعت پردازنده $\frac{90}{12} = 7.5$ برابر شده است، ولی سرعت اجرای کل برنامه $\frac{90+10}{12+10} = 4.5$ برابر خواهد شد. و زمان ورودی/خروجی برابر ۴۵٪ کل زمان اجرا خواهد بود.

۳- عوامل موثر بر کارایی یک سیستم ورودی/خروجی با توجه به پهنای باند و سرعت (تاخیر) را بیان کنید.

کارایی یک سیستم ورودی/خروجی، چه بر اساس پهنای باند و چه براساس سرعت به تمامی المان‌های بین دستگاه و حافظه بستگی خواهد داشت، شامل سیستم عامل که دستورهای ورودی/خروجی را تولید می‌کند. پهنای باند باس(گذرگاه ها)، حافظه، و میزان ماکسیمم انتقال از و یا به دستگاه ورودی/خروجی. مشابها سرعت به تاخیر دستگاه، به همراه تاخیر ایجاد شده در حافظه سیستم و گذرگاه‌ها. پهنای باند و سرعت موثر به سایر درخواست‌های ورودی/خروجی که ممکن است برای برخی منابع ایجاد اختلال کنند (مانند درخواست همزمان دو دستگاه از یک منبع) نیز بستگی دارد. در نهایت سیستم عامل گلوگاه خواهد بود؛ در برخی موارد سیستم عامل زمان زیادی را صرف می‌کند تا یک درخواست را از برنامه کاربر به یک ورودی/خروجی تحویل دهد که باعث ایجاد تاخیر زیاد می‌شود. در برخی موارد نیز سیستم عامل پهنای باند وروری/خروجی را به دلیل محدودیت‌های خود در پشتیبانی همزمان عمل‌های ورودی/خروجی، محدود می‌کند.

۴- می‌خواهیم ماکسیمم پهنای باند برای یک گذرگاه سنکرون و یک گذرگاه آسنکرون را با هم مقایسه کنیم. گذرگاه سنکرون سیکل ۵۰نانوثانیه دارد و هر انتقال یک سیکل طول می‌کشد. گذرگاه آسنکرون ۴۰ نانوثانیه برای هر handshake نیاز دارد. گذرگاه دیتا هر دو ۳۲ بیتی است. پهنای باند هر یک از گذرگاه‌ها را برای خواندن یک کلمه‌ای از یک حافظه ۲۰۰ نانوثانیه‌ای پیدا کنید.

زمان‌های مورد نیاز گذرگاه سنکرون مطابق زیر است:

فرستادن آدرس به حافظه: ۵۰ نانوثانیه

خواندن حافظه: ۲۰۰ نانوثانیه

فرستادن داده به دستگاه: ۵۰ نانوثانیه

در نتیجه ۳۰۰ نانو ثانیه نیاز خواهد داشت؛ یعنی حداکثر ۴بایت در هر ۳۰۰ نانوثانیه که برابر است با:

$$\frac{4}{300ns} = \frac{4MB}{0.3s} \approx 13.3 MB/s$$

در نگاه اول گذرگاه آسنکرون بسیار کندتر به نظر خواهد آمد، چرا که هفت مرحله نیاز دارد (با توجه به مراحل گفته شده در طول درس) که هر یک ۴۰ نانوثانیه زمان نیاز خواهند داشت و زمان مربوط به حافظه که ۲۰۰ نانوثانیه خواهد بود. ولی برخی از این مراحل همزمان انجام می‌شوند. حافظه آدرس را در پایان مرحله اول دریافت می‌کند و تا شروع مرحله پنجم نیازی به گذاشتن داده بر روی گذرگاه نخواهد بود؛ در نتیجه مراحل ۲و۳و۴ همزمان با زمان دسترسی حافظه انجام خواهند شد:

مرحله ۱: ۴۰ نانوثانیه

مراحل ۲، ۳، ۴: ۲۰۰ نانوثانیه(چراکه با زمان دسترسی حافظه همزمان انجام می‌شوند)

مراحل ۵، ۶، ۷: ۳*۴۰= ۱۲۰ نانوثانیه

در مجموع زمان مورد نیاز ۳۶۰ ثانیه خواهد بود:

$$\frac{4}{360ns} = \frac{4MB}{0.36s} \approx 111 MB/s$$

پس در نهایت گذرگاه سنکرون تنها حدود ۲۰٪ سریعتر خواهد بود.

۵- مزیت بزرگ وقفه نسبت به polling توانایی پردازنده برای انجام سایر کارها در حال انتظار برای یک وقفه است. فرض کنید پردازنده‌ای ۱گیگاهرتزی داریم که باید ۱۰۰۰ باید داده از یک ورودی بخواند. ورودی یک بایت در هر ۰. ۰۲ میلی ثانیه تولید می‌کند. کد پردازش داده و ذخیره آن در بافر ۱۰۰۰ سیکل طول می‌کشد.

الف- اگر پردازنده با polling متوجه حضور داده‌ای شود، و تکرار polling ۶۰ سیکل ظول بکشد، کل عملیات چند سیکل طول خواهد کشید؟

ب- اگر به جای polling از وقفه استفاده کنیم، چند سیکل از سایر عملیات می‌تواند هنگام ارتباط ورودی/خروجی صورت بگیرد؟(زمان اجرای وقفه ۲۰۰ سیکل است.)

الف- اگر فرض کنیم که پردازشگر داده را قبل از خواندن بایت بعدی پردازش می‌کند، سیکل‌های صرف poll شده برابر خواهند بود با:

0. 02 ms * 1 GHz - 1000 = 19,000

که برابر خواهد بود با: ۳۱۶. ۷ poll. از آنجا که کل زمان polling برای یافتن یک بایت نیاز است، تعداد سیکل‌های مصرف شده برای poll برابر خواهد بود با:

317 * 60 = 19,020

پس در نتیجه هر بایت ۱۹۰۲۰ + ۱۰۰۰= ۲۰۰۲۰ سیکل مصرف می‌کند. و کل عملیات

.۲۰۰۲۰ *۱۰۰۰ = ۲۰۰۲۰۰۰۰سیکل مصرف می‌کند

.ب- هر بار که بایتی وارد می‌شود، پردازنده ۲۰۰ + ۱۰۰۰=۱۲۰۰ سیکل برای پردازش داده مصرف می‌کند

0. 02 ms * 1 GHz - 1200 = 18800سیکل مصرف شده برای انجام سایر عملیات

در نتیجه کل سیکل مصرف شده برای انجام سایر عملیات برابر خواهد بود با:

18800 * 1000 = 18800000

بخش pipeline

۱. (کنکور ارشد ۱۳۸۰)در پردازنده‌ای با ساختار خط لوله (pipeline) دستورات هشت مرحله (stage) اجرا می‌شوند. چنانچه دستوری از نوع پرش (Branch) باشد به دستورات بعد اجازه‌ی ورود به خط لوله داده نمی‌شود تا این که اجرا دستور پرش به پایان برسد. برنامه‌ای درحال اجرا است که ۱۰۰ دستور دارد و بعد از هر ۱۹ دستور معمولی یک دستور پرش در آن ظاهر می‌شود. اگر تاخیر هر مرحله و ثبات‌های مربوط به آن جمعا 10ns باشد، اجرا این برنامه چقدر طول می‌کشد؟(برحسب ns)

۱) ۱۷۰۰ ۲) ۱۴۲۰ ۳) ۱۳۵۰ ۴) ۱۰۷۰

۲. برای اجرا دستورات در یک پردازنده باید ۴ کار متوالی انجام شود که مدت زمان هر کار به ترتیب 4ns، 4ns، 9ns، 3ns می‌باشد. اگر پردازنده‌ی دیگری طراحی کنیم که همین کارها را به صورت خط لوله (pipeline) انجام دهد و تاخیر ثبات خط لوله را 1ns فرض کنیم. افزایش سرعتی که پردازنده‌ی جدید در اجرا ۷ دستورالعمل متوالی غیر وابسته تولید می‌کند چقدر است؟

۱) ۱. ۴ ۲) ۲ ۳) ۱. ۷ ۴) ۴

۳. جدول زیر زمان صرف شده برای برخی عملیات در یک کامپیوتر را نشان می‌دهد. اگر زمان صرف شده برای عملیات ALU ۲۵٪ کاهش یابد.

الف) آیا این کاهش زمان‌ها در افزایش سرعت بدست آمده از تکنیک خط لوله تاثیر می‌گذارد؟اگر بله چقدر؟وگرنه چرا؟

ب) اگر این زمان‌ها ۲۵٪ افزایش یابد چطور؟

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

۴. در یک پردازنده دستورات به صورت خط لوله اجرا می‌شوند که دارای سه استگاه (stage) به قرار زیر است.

ایستگاه اول برای خواندن دستورات از حافظه، ایستگاه دوم برای اجرا دستورات، ایستگاه سوم برای ذخیره حاصل در حافظه یا بارگیری از حافظه.

دو روش پیاده‌سازی را باهم مقایسه می‌کنیم: روش اول استفاده از یک حافظه‌ی یکپارچه است و روش دو بهره گرفتن از یک حافظه برای دستورات و یک حافظه برای داده می‌باشد.

اگر قطعه برنامه‌ای دارای ۱۰۰ دستور باشد که همگی رجوع به حافظه دارند و هیچ نوع وابستگی داده بین آنها هم نیست آنگاه نسبت زمان اجرا روش اول را به روش دوم حساب کنید.

۱) 100/102 ۲) 200/102 ۳) 300/102 ۴) 204/102

۵. (stallings 6thed- 12. 3)اگر n تعداد دستورهای اجرایی، p احتمال این که به یک دستور پرش شرطی یا غیر شرطی برخورد کنیم و q احتمال این که اجرای یک دستور پرش باعث پریدن به یک آدرس غیر متوالی شود. و با فرض این که اجرای چنین دستوری نیاز به خالی شدن تمام مراحل خط لوله داشته باشد. زمان اجرای این n دستورالعمل با این تکنیک خط لوله چقدر است؟

۶. (کنکور ارشد) اگر یک خط لوله (pipeline) سه ایستگاهی را به چهار ایستگاه تبدیل کنیم پریود ساعت از T به 0. 9T کاهش میابد فرض کنید ۳۰٪ دستورات پرش هستند. دستور بعد از دستور پرش وارد لوله نمی‌شود تا اینکه دستور پرش به اتمام برسد. نسبت زمان اجرا n دستور در ساختار سه ایستگاهی به ساختار چهار ایستگاهی چقدر است؟

حل مسائل:

در مورد سوالات خط لوله دو فرمول زیر مطرح است:

1. $T_k=[k+(n-1)]\tau$

2. $T_1=nk\ \tau$

که τ طول زمان اجرای هر مرحله (stage) از خط لوله، k تعداد مراحل(stage)، n تعداد دستورالعمل‌ها، T_k زمان اجرا با تکنیک خط لوله و T_1 زمان اجرا به صورت عادی است.

۱. برای اجرای اولین دستور این برنامه 80ns وقت لازم است و بعد از آن تا ۱۸ دستور قبل از دستور پرش هر کدام 10ns طول می‌کشند. برای دستور پرش بعدی باز 80ns وقت لازم است و برای ۱۹ دستور بعد از پرش اول باز هر کدام 10ns زمان میخواهند. به همین ترتیب داریم:

t= (8+18)*10 + ((8+19)*10)*3 + (8+20)*10 = 1350ns

۲. اگر این ۷ دستورالعمل را بدون تکنیک خط لوله انجام دهیم زمان 7*(3+9+4+4)ns=140ns صرف می‌شود. حال اگر از تکنیک خط لوله استفاده کنیم باید مدت زمان هر مرحله را برابر ۹ یعنی ماکزیمم زمان مراحل در نظر بگیریم و با توجه به این که تاخیر ثبات خط لوله ۱ است پس تاخیر هر مرحله از خط لوله برابر 10ns است و باتوجه به این که دستورالعمل‌ها به هم وابسطه نیستند مدت زمان صرف شده برابر خواهد بود با: (4+6)*10=100ns با این حساب افزایش سرعت برابر با ۱. ۴ برابر خواهد بود.

.۳

الف)کاهش زمان انجام عملیات در ALU تاثیری در افزایش سرعت حاصل از تکنیک خط لوله نمی‌گذارد. زیرا این کار تاثیری بر طول چرخه‌ی کلاک نمی‌گذارد.

ب)اگر زمان عملیات ALU ۲۵٪ بیشتر شود این عملیات گلوگاه عملیات خط لوله محسوب خواهند شد. و در این حالت طول چرخه‌ی کلاک باید 250ns بشود. در این صورت سرعت ۲۰٪ کاهش میابد.

۴. اگر از دو حافظه یکی برای دستورات و یکی برای داده استفاده کنیم حالت بهینه برای خط لوله است و زمان اجرای ۱۰۰ دستورالعمل برابر خواهد بود با: τ=102 τ *(3+99) ولی اگر از یک حافظه‌ی یکپارچه استفاده کنیم چون نمیتوان به طور موازی هم از حافظه دستورالعمل واکشی کرد و هم داده پس در فرایند خط لوله بعد

از وارد شدن دو دستور اول دو حباب نیز وارد می‌شود تا خواندن از حافظه تداخل پیدا نکند پس زمان کل در این صورت برابر است با. 200τ

۵. تعداد دستوراتی که باعث این نوع پرش می‌شوند pqn و تعداد دستوراتی که باعث این نوع پرش نمی‌شوند (1-pq)n است. با استفاده از فرمول شماره ۱ و ۲ داریم: $T_k=pqnk\tau + (1-pq)[k+n-1]\ \tau$

۶. با توجه به فرمول بدست آمده از سوال قبل داریم:

$$\frac{1.6n + 1.4}{1.71n + 1.89}$$

فصل ۵ – سازمان و طراحی یک کامپیوتر پایه

۱- یک کامپیوتر از حافظه‌ای با 256k کلمه ۳۲ بیتی استفاده می‌کند. یک دستورالعمل دودویی در یک کلمه از حافظه ذخیره شده است. دستورالعمل چهار بخش دارد: بیت غیر مستقیم، یک کد عملیاتی، یک کد ثبات برای تعیین یکی از ۶۴ ثبات و بخش آدرس.

الف) چند بیت در کدعملیاتی، کد ثبات و آدرس وجود دارد؟

ب) قالب کلمه دستورالعمل را ترسیم و تعداد بیت در هر قسمت را معین کنید.

پ) در ورودی‌های داده و آدرس حافظه چند بیت وجود دارد؟

۲- اختلاف بین دستور با آدرس مستقیم و غیرمستقیم چیست؟ چند ارجاع به حافظه برای هر نوع دستورالعمل لازم است تا عملوند را به ثبات پردازشگر منتقل کند

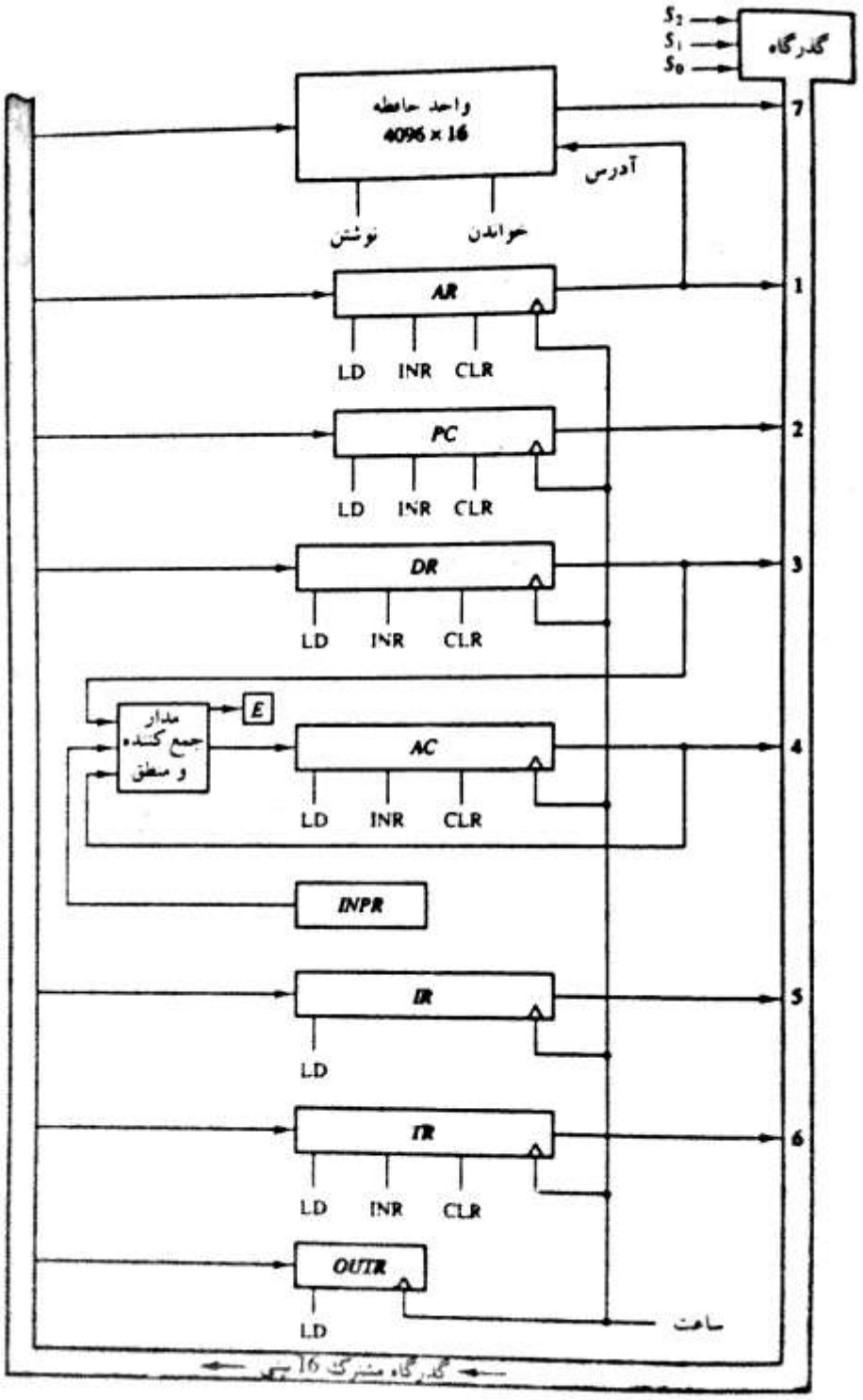
۳- ورودی‌های کنترل زیر در سیستم گذرگاه شکل ۴-۵ فعالند. برای هر حالت انتقال ثباتی که در پالس ساعت بعدی اجرا شود را معین کنید.

	S_2	S_1	S_0	ثبات LD	حافظه	جمع‌کننده
الف)	1	1	1	*IR*	خواندن	—
ب)	1	1	0	*PC*	—	—
ج)	1	0	0	*DR*	نوشتن	—
د)	0	0	0	*AC*	—	جمع

۴- انتقال ثبات‌های زیر قرار است در سیستم شکل ۴-۵ اجرا شوند. برای هر انتقال: (۱)مقدار دودویی که باید به ورودی‌های انتخاب گذرگاه S_2، S_1، S_0 اعمال شوند را معین کنید؛ (۲)ثباتی که کنترل LD آن باید فعال شود(اگر وجود دارد)؛ (۳)عمل نوشتن یا خواندن حافظه(اگر لازم است)؛ و (۴)عمل در جمع کننده و مدار منطقی (اگر وجود دارد).

الف) $AR \leftarrow PC$ ب) $IR \leftarrow M[AR]$

ج) M[AR] ← TR د) AC ← DR, DR ←AC



شکل ۴-۵ ثبات‌های کامپیوتر پایه متصل به یک گذرگاه مشترک

۵- توضیح دهید چرا هیچیک از ریزعمل‌های زیر در طول یک پالس ساعت در سیستم شکل ۴-۵ اجرا نمی‌شوند. رشته ریزاعمال لازم برای انجام عملیات را معین کنید

الف) IR ← M[PC]

ب) AC ← AC + TR

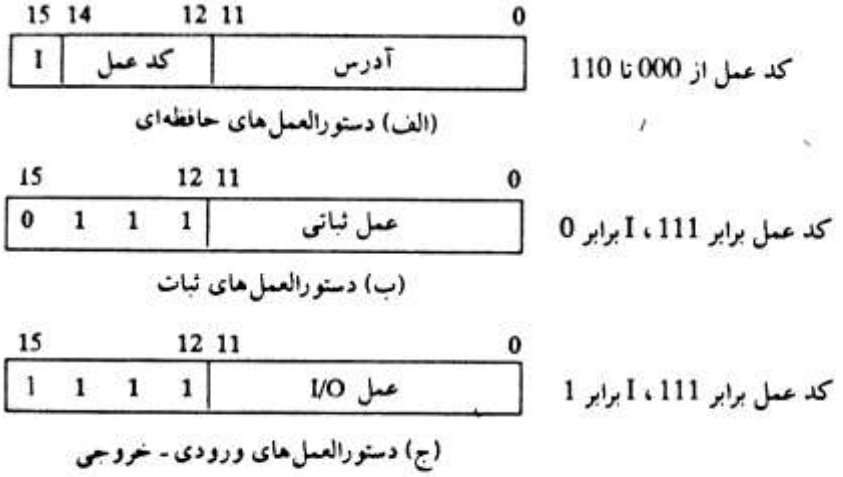
ج) (AC تغییر نمی‌کند) DR ← DR + AC

۶- قالب دستورات کامپیوتر پایه شکل ۵-۵ و لیست دستورات جدول ۲-۵ را ملاحظه کنید. برای هر یک از دستورا ۱۶ بیتی، کد معادل چهاربیتی مبنای شانزده را نوشته و بزبان ساده بگوئید این دستور چه کاری انجام می‌دهد.

الف) ۰۱۰۰ ۰۰۱۰ ۰۰۰۰ ۰۰۰۱

ب) ۰۱۰۰ ۰۰۱۰ ۰۰۰۱ ۱۰۱۱

ج) ۰۱۱۱ ۰۰۰۰ ۰۰۱۰ ۰۰۰۰



شکل ۵-۵ قالب دستورالعمل‌ها در کامپیوتر پایه

جدول ۲-۵ دستورالعمل‌های کامپیوتر پایه

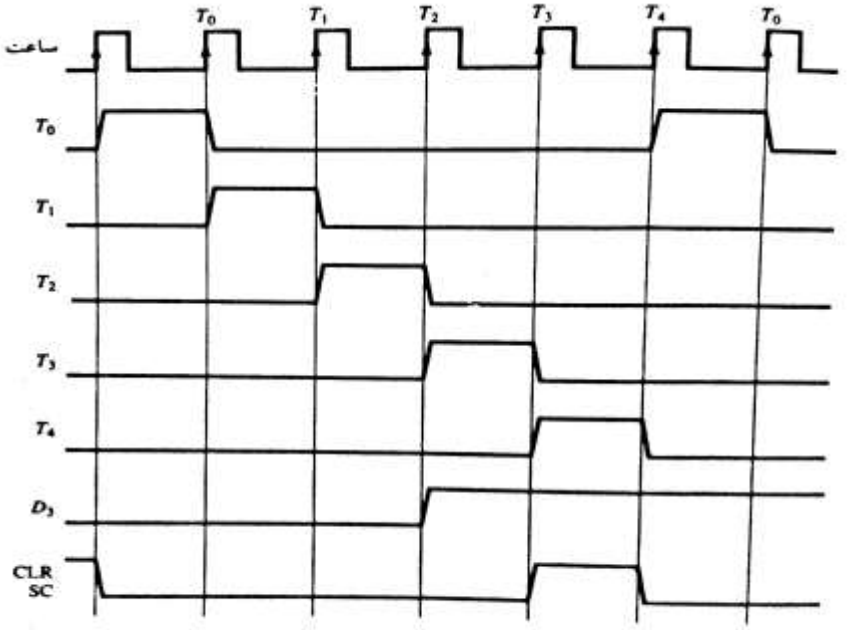
سمبل	کد شانزده شانزدهی $I = 0$	کد شانزده شانزدهی $I = 1$	شرح
AND	0xxx	8xxx	AND کردن کلمه حافظه با AC
ADD	1xxx	9xxx	جمع کردن کلمه حافظه با AC
LDA	2xxx	Axxx	بار کردن کلمه حافظه در AC
STA	3xxx	Bxxx	ذخیره محتوای AC در حافظه
BUN	4xxx	Cxxx	انشعاب نامشروط
BSA	5xxx	Dxxx	انشعاب و ضبط آدرس بازگشت
ISZ	6xxx	Exxx	افزایش و گذر در صورت نتیجه صفر
CLA	7800		پاک کردن AC
CLE	7400		پاک کردن E
CMA	7200		متمم کردن AC
CME	7100		متمم کردن E
CIR	7080		چرخش AC و E به راست
CIL	7040		چرخش AC و E به چپ
INC	7020		افزایش AC
SPA	7010		گذر از دستور بعدی اگر AC مثبت باشد
SNA	7008		گذر از دستور بعدی اگر AC منفی باشد
SZA	7004		گذر از دستور بعدی اگر AC صفر باشد
SZE	7002		گذر از دستور بعدی اگر E صفر باشد
HLT	7001		توقف کامپیوتر
INP	F800		دریافت کاراکتر و انتقال آن به AC
OUT	F400		برداشتن کاراکتر از AC و انتقال آن به خروجی
SKI	F200		گذر مبتنی بر پرچم ورودی
SKO	F100		گذر مبتنی بر پرچم خروجی
ION	F080		فعال کردن وقفه‌ها
IOF	F040		غیرفعال کردن وقفه‌ها

۷- کدام دو دستور برای ۱ کردن فلیپ فلاپ E در کامپیوتر پایه بکار می‌روند؟

۸- یک دیاگرام زمانبندی مشابه شکل ۷-۵ ترسیم و فرض کنید SC در T_3 برابر ۰ شده باشد بشرط اینکه سیگنال کنترل C_7 فعال باشد.

C_7T_3: SC ← 0

C_7 با لبه پالس مربوط به T_1 فعال می‌شود.



شکل ۷-۵ مثالی از سیگنال‌های زمانبندی واحد کنترل

۹- محتویات AC در کامپیوتر پایه عدد مبنای شانزده A937 است و مقدار اولیه E برابر ۱ است. محتویات AC، E، PC، AR و IR در مبنای ۱۶ پس از اجرای دستور CLA چیست. عمل قبل را ۱۱ بار با هر یک از دستورالعمل‌ها تکرار کنید. مقدار اولیه PC را عدد مبنای شانزده 021 فرض کنید.

۱۰- دستورالعملی در آدرس ۰۲۱ کامپیوتر پایه دارای I=0، کد عملیاتی AND و آدرس ۰۸۳ است(تمام ارقام در مبنای شانزده است). کلمه حافظه واقع در آدرس ۰۸۳ دارای عملوند B8F2 و محتویات AC هم A937 است. در طول سیکل دستور محتویات ثبات‌های زیر را در پایان فا اجرا معین کنید: PC، AR، DR، AC و IR. مسئله را شش بار دیگر برا دستورالعمل ارجاع به حافظه دیگری تکرار کنید.

۱۱- محتویات ثبات‌های PC، AR، DR، IR و SC در مبنای شانزده وقتی که در کامپیوتر پایه دستور غیر مستقیم ISZ دریافت و اجرا شود چیست. مقدار اولیه PC را 7FF در نظر بگیرید. محتویات حافظه در آدرس 7FF برابر EA9F است. محتویات حافظه در آدرس A9F هم 0C35 می‌باشد. محتویات حافظه C35 برابر با FFFF می‌باشد. پاسخ خود را بصورت جدولی با پنج ستون با هر ستون برای یک ثبات، و هر سطر برای یک سیگنال زمان بندی تهیه کنید. محتویات هر ثبات را پس از لبه مثبت هر پالس ساعت نشان دهید.

۱۲- محتویات PC در کامپیوتر پایه 3AF است (تمام اعداد در مبنای شانزده). محتویات AC هم 7EC3 است. محتویات حافظه آدرس 3AF برابر با 932E می‌باشد. محتویات حافظه در آدرس 32E برابر 09AC و در آدرس 9AC هم 8B9F است

الف) دستورالعملی که بعدا دریافت و اجرا شود چیست؟

ب) عمل دودویی که در AC پس از اجرای دستورالعمل رخ می‌دهد چیست؟

ج) محتویات ثبات‌های PC، AR، DR، AC و IR در مبنای شانزده چیست، همچنین مقادیر E و I و SC در انتهای سیکل دستورالعمل را معین کنید.

۱۳- فرض کنید که شش دستور ارجاع به حافظه در کامپیوتر پایه در جدول ۴-۵ با جدول زیر تعویض شوند. EAآدرس موثر واقع در AR در T_4 است. فرض کنید که جمع کننده و مدار منطقی شکل ۴-۵ می‌تواند عمل XOR را انجام دهد AC XOR DR←AC بعلاوه فرض کنید که جمع کننده و مدار منطقی نمی‌توانند مستقیما تفریق را انجام دهند. تفریق باید بکمک مکمل ۲ انجام شود. رشته عبارات انتقال ثبات لازم برای اجرای هر دستور لیست شده را از T_4 به بعد مشخص کنید. دقت کنید که هیچ تغییری در AC رخ نمی‌دهد مگر اینکه دستورالعمل تغییری را در آن معین کند. شما می‌توانید با استفاده از TR محتویات AC را موقتا ذخیره و یا محتویات AC و DR را با هم عوض کنید.

سمبل	کد عمل	نمایش سمبلیک	توضیح
XOR	000	$AC \leftarrow AC \oplus M[EA]$	OR انحصاری با AC
ADM	001	$M[EA] \leftarrow M[EA] + AC$	جمع AC با حافظه
SUB	010	$AC \leftarrow AC - M[EA]$	تفریق حافظه از AC
XCH	011	$AC \leftarrow M[EA],\ M[EA] \leftarrow AC$	تبادل AC با حافظه
SEQ	100	If $(M[EA] = AC)$ then $(PC \leftarrow PC + 1)$	گذر در صورت برابری
BPA	101	If $(AC > 0)$ then $(PC \leftarrow EA)$	انشعاب اگر AC مثبت و غیر صفر باشد

۱۴- تغییرات زیر را در کامپیوتر پایه بعمل آورید.

۱- یک ثبات CTR (ثبات شمارنده) را به سیستم گذرگاه اضافه کنید و آنرا با $S_2S_1S_0$= 000 انتخاب نمایید.

۲- ISZ را با دستوری که یک عدد را در CTR

LDC Address CTR ← M[address]

۳- یک دستور ارجاع به ثبات ICSZ به مجموعه اضافه کنید: CTR را یک واحد اضافه کرده و از اجرای دستور بعدی اگر حاصل افزایش صفر است صرفنظر نمائید. مزیت این تغییر را بیان کنید.

۱۵- واحد حافظه کامپیوتر پایه در شکل ۱۳-۵ را با یک حافظه ۱۶ × ۶۵۵۳۶ عوض کنید. ان حافظه آدرس ۱۶ بیتی نیاز دارد. قالب دستورالعمل ارجاع به حافظه در شکل(۵-۵-الف) برای I=1 بلاتغییر و بخش آدرس در مکان‌های ۰ تا ۱۱ قرار دارد. اما وقتی I=0 است (آدرس مستقیم) آدرس دستورالعمل توسط ۱۶ بیت در کلمه دیگری که بدنبال دستور آمده است داده شده است. ریز اعمال را در T_2، T_3 (و T_4 اگر لازم باشد) تصحیح کنید تا با این پیکر بندی همانگ باشد.

۱۶- کامپیوتری از یک حافظه هشت بیتی ۶۵۵۳۶ کلمه‌ای استفاده می‌کند. این کامپیوتر دارای ثبات‌های PC، AR، TR (هر یک شانزده بیت) و AC و DR و IR (هر یک هشت بیت) است. یک دستور ارجاع به حافظه متشکل از سه کلمه است: یک کد عملیات ۸ بیتی (یک کلمه) و یک

آدرس ۱۶ بیتی (در دو کلمه بعدی). تمام عملوندها هشت بیت هستند. بیت غیر مستقیم هم وجود ندارد.

الف) بلاک دیاگرامی از کامپیوتر ترسیم کنید و ثبات‌ها و حافظه‌ها را مطابق شکل ۳-۵ نشان دهید. (از یک گذرگاه مشترک استفاده نکنید).

ب) طرز قرار گرفتن یک نمونه دستور سه کلمه‌ای را به همراه عملوند ۸ بیتی در حافظه نشان دهید.

رشته ریز اعمال برای دریافت یک دستور ارجاع به حافظه را لیست کنید و سپس عملوند را در DR قرار دهید. از سیگنال زمانی T_0 شروع کنید.

۱۷- یک کامپیوتر دیجیتال دارای ۱۶۳۸۴ حافظه ۴۰ بیتی در هر کلمه است. قالب کد دستور از شش بیت برای عملوند و ۱۴ بیت برای آدرس تشکیل شده است. (بیت غیر مستقیم ندارد). دو دستورالعمل دی یک کلمه جای داده شده ان و یک ثبات دستورالعمل ۴۰ بیتی IR هم در واحد کنترل وجود دارد. برنامه‌ای را برای فازهای برداشت و اجرا در این کامپیوتر بنویسید.

۱۸- یک برنامه خروجی از آدرس ۲۳۰۰ نوشته شده است. این برنامه وقتی کامپیوتر یک وقفه را در FGO = 1 تشخیص دهد اجرا می‌گردد (در حالیکه IEN = 1 است)

الف- چه دستوری باید در آدرس ۱ قرار گیرد؟

ب- دو دستور آخر برنامه خروجی چیست؟

۱۹- عبارات انتقال ثبات برای ثبات R و حافظه در یک کامپیوتر مطابق زیر است (Xها توابع کنترل هستند و بطور تصادفی رخ می‌دهند).

کلمه حافظه را در R بنویس $X_3\ X_1: R \leftarrow M[AR]$

انتقال AC به R $X_1\ X_2: R \leftarrow AC$

R را در حافظه بنویس $X1\,X_3: M[AR] \leftarrow R$

حافظه دارای ورودی‌های داده، خروجی‌های داده، ورودی‌های آدرس و ورودی‌های کنترل برای خواندن و نوشتن مطابق شکل ۱۲-۲ است. سخت افزار R و حافظه را بشکل بلاک دیاگرام بکشید. نشان دهید که چگونه توابع کنترل X_1 تا X_3 ورودی R، ورودی مولتی پلکسرهائیی که شما در دیاگرام وارد کرده اید، و ورودی‌های خواندن و نوشتن حافظه را انتخاب می‌کنند.

۲۰- رشته اعمالی که باید توسط فلیپ فلاپ F انجام شوند با عبارت انتقال ثبات داده شده اند

۱ در F نشانده شود	$XT_3: F \leftarrow 1$
F با ۰ پاک شود	$yT_1: F \leftarrow 0$
مکمل F	$ZT_2: F \leftarrow F'$
مقدار G را به F انتقال بده	$WT_5: F \leftarrow G$

در غیر اینصورت F نباید تغییر یابد. دیاگرام منطقی مربوط به اتصالات گیت‌ها که توابع کنترل و ورودی‌های فلیپ فلاپ F تشکیل می‌دهند را ترسیم نمائید. از یک فلیپ فلاپ JK استفاده کرده و تعداد گیت‌ها را حداقل کنید.

۲۱- مدار کنترل گیتی مربوط به شمارنده برنامه PC را در کامپیوتر پایه بدست آورید.

۲۲- مدار کنترل گیتی برای ورودی خواندن یک حافظه را در کامپیوتر پایه بدست آورید.

۲۳- مدار کامل منطقی وقفه را در کامپیوتر پایه نشان دهید. از فلیپ فلاپ JK استفاده کرده و گیت‌ها را به حداقل برسانید.

۲۴- عبارت بولی را برای X_2 (جدول ۷-۵ را ملاحظه کنید) بدست آورید. نشان دهید که X_2 می‌تواند با یک گیت AND و یک OR تولید شود.

۲۵- عبارت بولی برای یک مدار گیتی که شمارنده SC را پاک کند بدست آورید. دیاگرام منطقی آنرا رسم نموده و نشان دهید که چگونه خروجی به ورودی‌های INR و CLR از شمارنده SC وصل می‌شود شکل ۶-۵. تعداد گیت‌ها را حداقل نمائید.

حل مسائل فصل ۵ معماری – طراحی کامپیوتر پایه

۱- $256K = 2^8 \times 2^{10} = 2^{18}$

$64 = 2^6$

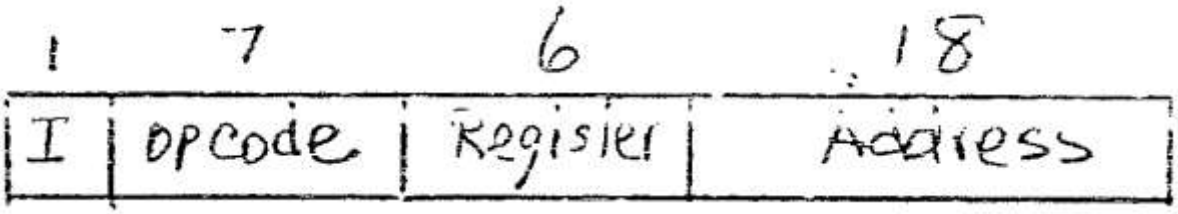
الف) آدرس: ۱۸ بیت

کد ثبات: ۶ بیت

بیت غیر مستقیم: ۱ بیت مجموع: ۲۵ بیت

بیت عملیاتی: $32 - 25 = 7$

ب) ۳۲ بیت



پ) داده ۳۲ بیت آدرس ۱۸ بیت

۲- دستور با آدرس مستقیم ۲ ارجاع لازم دارد ۱) خواندن دستور ۲) خواندن عملوند

اما دستور با آدرس غیرمستقیم ۳ ارجاع لازم دارد ۱)خواندن دستور ۲) خواندن آدرس عملوند ۳) خواندن عملوند

۳-

۱) داده از حافظه خوانده می‌شود روی گذرگاه قرار می‌گیرد و در IR بارگذاری می‌شود. IR

← M[AR]

۲) TR به گذرگاه و بعد در PC بار گذاری می‌شود: PC ← TR

۳) AC به گذرگاه، نوشتن در حافظه و بارگذاری در DR :

AC ← M[AR] و DR ← AC

۴) جمع DR (یا INPR) با AC نتیجه به AC : AC ← AC + DR

-۴

		Adder	memory	load	$S_2S_1S_0$	
۱	AR ← PC	_	_	AR	010(PC)	AR ← PC
۲	IR ← M[AR]	_	Read	TR	111(M)	IR ← M[AR]
۳	M[AR] ← TR	_	Write	_	110(TR)	M[AR] ← TR
۴	DR ← AC DR ← AC	انتقال DR به AC	_	DC, AC	100(AC)	DR ← AC

-۵

الف) IR ← M[PC]

PC نمی‌تواند آدرس را برای حافظه فراهم کند. آدرس باید ابتدا به AR منتقل شود

AR ← PC

IR ← M[AR]

ب) AC ← AC + TR

عمل جمع باید با DR انجام شود، TR باید ابتدا به DR منتقل شود

$DR \leftarrow TR$

$AC \leftarrow AC + DR$

ج) $DR \leftarrow DR + AC$

جواب عمل جمع به AC منتقل می‌شود(نه DR) برای اینکه مقدار AC تغییر نکند مقدار آن باید ابتدا در DR (یا TR) نگه داری شود. (به جواب ۴ نگاه کنید)

$AC \leftarrow DR;\ DR \leftarrow AC$

$AC \leftarrow AC + DR$

$AC \leftarrow DR;\ DR \leftarrow AC$

۶-

الف) $(1024)_{16} = 0001\ 0000\ 0010\ 0100$

ADD 024: محتوای M[024] را با AC جمع می‌کند و در AC ذخیره می‌کند.

ب) $(B124)_{16} = 1011\ 0001\ 0010\ 0100$

STAI 124: AC را در خانه‌ای از حافظه ذخیره می‌کند که آدرس آن محتوای M[124] است

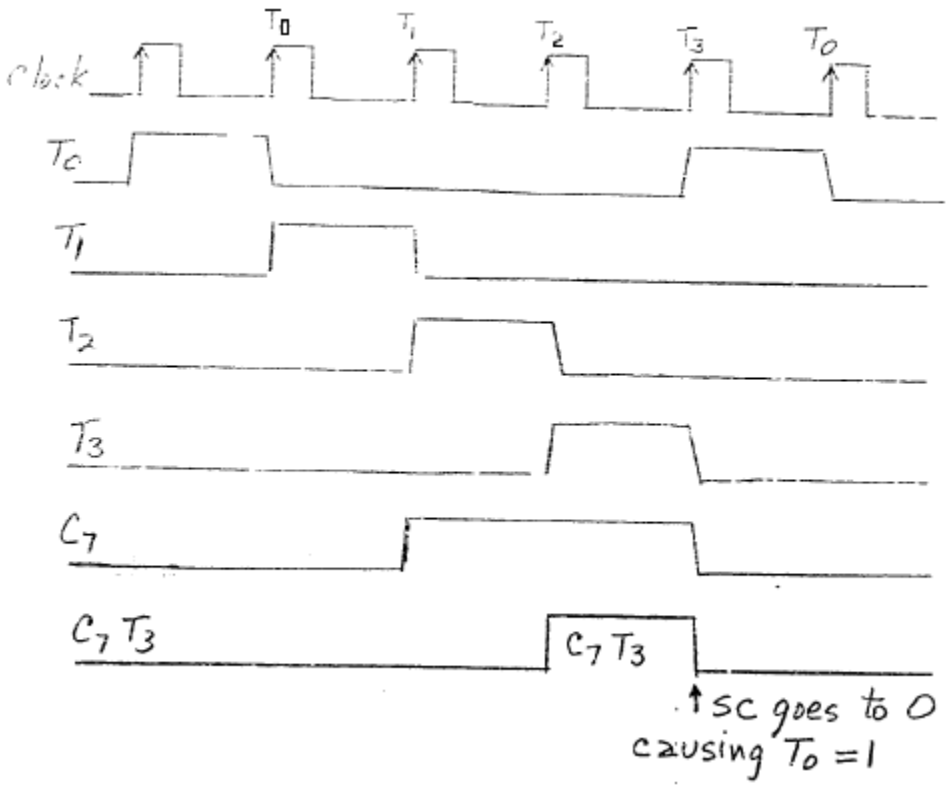
پ) $(7020)_{16} = 0111\ 0000\ 0010\ 0000$

INC، به AC یک واحد اضافه می‌کند

۷- ClearE $\equiv$ CLE

complementE $\equiv$ CME

-۸



-۹

	E	AC	PC	AR	IR
Initial	1	A937	021	—	—
CLA	1	0000	022	800	7800
CLE	0	A937	022	400	7400
CMA	1	56C8	022	200	7200
CME	0	A937	022	100	7100
CIR	1	D49B	022	080	7080
CIL	1	526F	022	040	7040
INC	1	A938	022	020	7020
SPA	1	A937	022	010	7010
SNA	1	A937	023	008	7008
SZA	1	A937	022	004	7004
SZE	1	A937	022	002	7002
HLT	1	A937	022	001	7001

-۱۰

	PC	AR	DR	AC	IR
Initial	021	—	—	A937	—
AND	022	083	B8F2	A832	0083
ADD	022	083	B8F2	6229	1083
LDA	022	083	B8F2	B8F2	2083
STA	022	083	—	A937	3083
BUN	083	083	—	A937	4083
BSA	084	084	-	A937	5083
ISZ	022	083	B8F3	A937	6083

-۱۱

	PC	AR	DR	IR	SC
Initial	7FF	—	—	—	0
T_0	7FF	7FF	—	—	1
T_1	800	7FF	—	EA9F	2
T_2	800	A9F	—	EA9F	3
T_3	800	C35	—	EA9F	4
T_4	800	C35	FFFF	EA9F	5
T_5	800	C35	0000	EA9F	6
T_6	801	C35	0000	EA9F	0

-۱۲

الف) $(1001)_2 = 9$

ADD I 323 $\quad$ I = 1 و ADD ← 001

Memory	
3AF	932E
32E	09AC
9AC	8B9F

AC= 7EC3

ب) AC = 7EC3

DR = 8B9F (ADD)

result = 0A62

E = 1

ج) PC = 3AF + 1 = 3B0 IR = 932E

AR = 9AC E = 1

DR = 889F I = 1

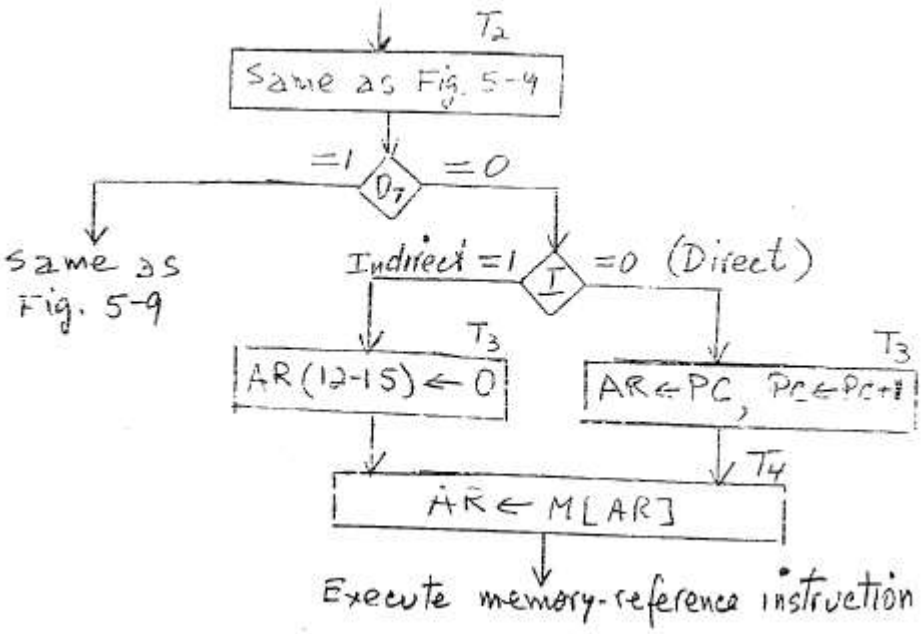
AC = 0A62 SC = 0000

-۱۳

XOR	D_0T_4 :	$DR \leftarrow M[AR]$
	D_0T_5 :	$AC \leftarrow AC \oplus DR,\ SC \leftarrow 0$
ADM	D_1T_4 :	$DR \leftarrow M[AR]$
	D_1T_5 :	$DR \leftarrow AC,\ AC \leftarrow AC + DR$
	D_1T_6 :	$M[AR] \leftarrow AC,\ AC \leftarrow DR,\ SC \leftarrow 0$
SUB	D_2T_4 :	$DR \leftarrow M[AR]$
	D_2T_5 :	$DR \leftarrow AC,\ AC \leftarrow DR$
	D_2T_6 :	$AC \leftarrow \overline{AC}$
	D_2T_7 :	$AC \leftarrow AC + 1$
	D_2T_8 :	$AC \leftarrow AC + DR,\ SC \leftarrow 0$
XCH	D_3T_4 :	$DR \leftarrow M[AR]$
	D_3T_5 :	$M[AR] \leftarrow AC,\ AC \leftarrow DR,\ SC \leftarrow 0$
SEQ	D_4T_4 :	$DR \leftarrow M[AR]$
	D_4T_5 :	$TR \leftarrow AC,\ AC \leftarrow AC \oplus DR$
	D_4T_6 :	$If\ (AC=0)\ then\ (PC \leftarrow PC+1),\ AC \leftarrow TR,\ SC \leftarrow 0$
BPA	D_5T_4 :	$If\ (AC=0 \wedge AC(15)=0)\ then\ (PC \leftarrow AR),\ SC \leftarrow 0$

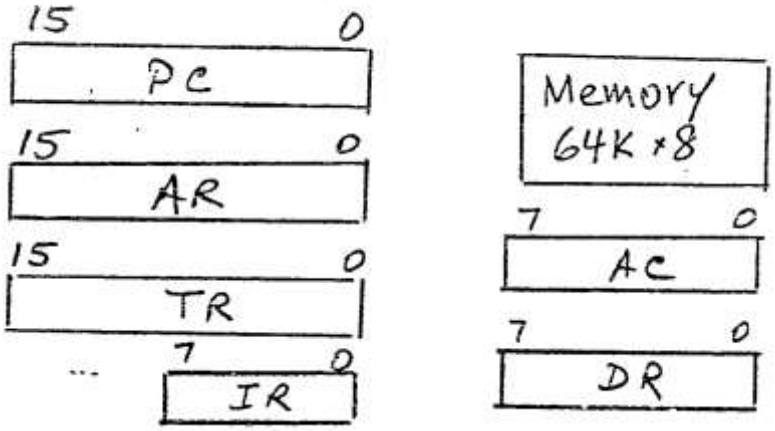
-۱۴ این تغییرات باعث می‌شوند که دستور ISZ به جای ۲ بار ارجاع به حافظه، ۲ بار به ثبات‌ها رجوع می‌کند، دستور جدید (ICSZ) می‌تواند در زمان T_3 به جای T_6 انجام شود که این باعث صرفه جویی ۳ پالس ساعت در اجرای دستورالعمل می‌شود

-۱۵

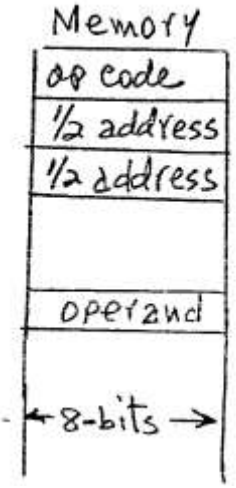

T2
Same as Fig. 5-4
=1
D7
=0
Same as Fig. 5-9
Indirect =1
I
=0 (Direct)
T3
AR(12-15) ← 0
T3
AR ← PC, PC ← PC+1
T4
AR ← M[AR]
Execute memory-reference instruction

-۱۶

الف)


15
0
PC
15
0
AR
15
0
TR
7
0
IR
Memory
64K x 8
7
0
AC
7
0
DR

ب)



ج)

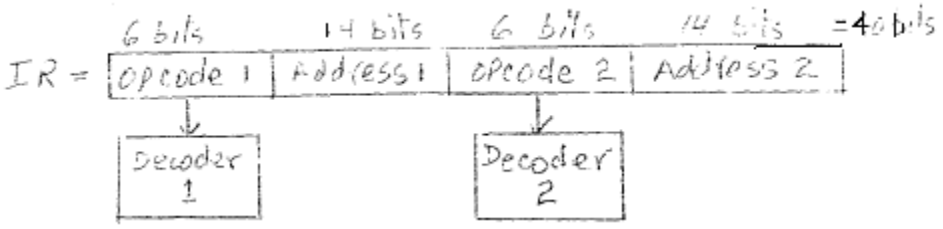
T_0: IR ← M[PC], PC ← PC+1

T_1: AR(0-7) ← M[PC], PC ← PC+1

T_2: AR(8-15) ← M[PC], PC ← PC+1

T_3: DR ← M[AR]

-۱۷



۱) ۴۰ بیت (دو دستور) را در حافظه خوانده و در PC بارگذاری می‌کنیم و PC رایک واحد افزایش می‌دهیم

۲) کد عملیاتی ۱ را رمزگشایی می‌کنیم

۳) دستور ۱ را با استفاده از آدرس ۱ انجام می‌دهیم

۴) کد عملیاتی ۲ را رمز گشایی می‌کنیم

۵) دستور ۲ را با استفاده از آدرس ۲ انجام می‌دهیم

۶) به مرحله‌ی ۱ بازمی گردیم

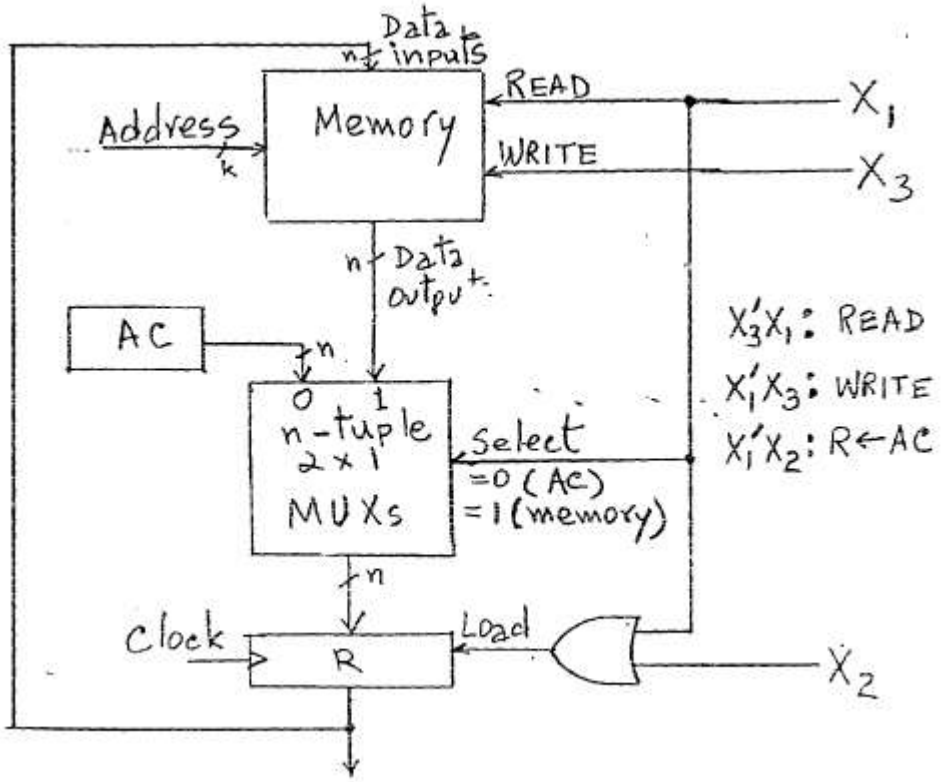
-۱۸

الف) BUN 2300

ب) ION

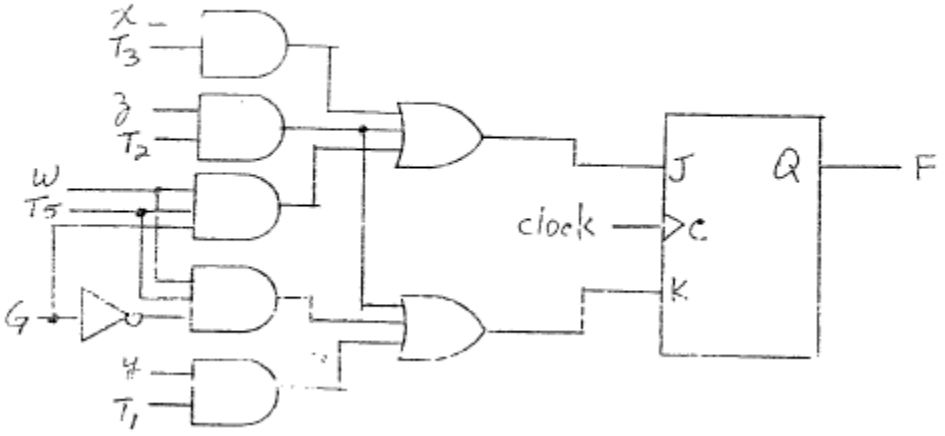
BUN 0 I

-۱۹



-۲۰

$J_F = x T_3 + z T_2 + w T_5 G \qquad K_F = y T_1 + z T_2 + w T_5 G'$



-۲۱ از شکل ۶-۵ داریم:

$Z_{DR} = 1$ if $DR = 0$; $Z_{AC} = 1$ if $AC = 0$

$$INR(PC) = R'T_1 + RT_2 + D_6T_6Z_{DR} + pB_9(FGI) + pB_8(FGO) + rB_4(AC_{15})' + rB_3(AC_{15}) + rB_2Z_{AC} + rB_1E'$$

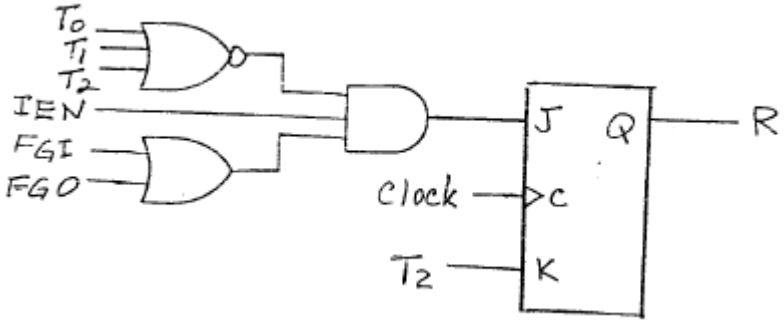
$$LD(PC) = D_4T_4 + D_5T_5$$

$$CLR(PC) = RT_1$$

-۲۲ $Write = D_3T_4 + D_5T_4 + D_6T_6 + RT_1\ (M[AR] \leftarrow xx)$

-۲۳ $(T_0 + T_1 + T_2)'(IEN)(FGI + FGO): R \leftarrow 1$

$RT_2: R \leftarrow 0$

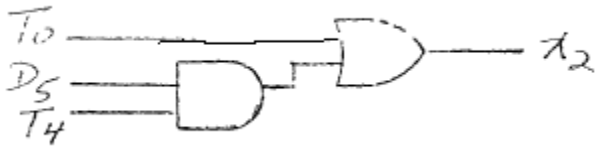


-۲۴ $R'T_0: AR \leftarrow PC$

$RT_0: TR \leftarrow PC$

D_5T_4: M[AR] ← PC

$X_2 = R'T_0 + RT_0 + D_5T_4 = (R'+R)T_0 + D_5T_4 = T_0 + D_5T_4$



۲۵- $CLR(SC) = RT_2 + D_7T_3(I'+I) + (D_0+D_1+D_2+D_5)T_5 + (D_3+D_4)T_4 + D_6T_6$