Subject:

Year. Month. Date. ( )

مثال: در مثال قبل clock-rate برابر 500MHz است، execution time و MIPS را برای هر برنامه بدست آورید.

$$MIPS = \frac{IC}{Execution\ time \times 10^6} = \frac{f}{CPI \times 10^6}$$

	CPI
A	1
B	2
C	3

	A	B	C
1	5	1	1
2	10	1	1

program1:

$$CPI = \frac{1}{7}(5\times 1 + 2 + 3) = \frac{10}{7}$$

$$MIPS = \frac{500 \times 10^6}{\frac{10}{7} \times 10^6} = 350$$

$$Execution\ time = \frac{7 \times 10}{7 \times 500 \times 10^6} = 20\,ns$$

program2:

$$Exec_2 = \frac{10 + 2 + 3}{500 \times 10^6} = 30\,ns$$

$$MIPS = \frac{12}{30 \times 10^{-9} \times 10^{6}} = 400$$

برای مشخص کردن کدام سیستم بهتر است Exect را بررسی کنیم این کار را انجام می‌دهیم چون مقدار عملی را نشان می‌دهد ولی MIPS نشان می‌دهد برنامه دوم بهتر است اجرای این کد در دستور داده ولی ممکن است تعداد دستورها تبدیل سیکل‌ها تفاوت داشته باشد

Subject:

Year. Month. Date. ( )

8 Performance Evaluation

گروهی به نام SPEC وجود دارد که هر سال ارزیابی می‌کند کامپایلرها از چه برنامه‌هایی بیشتر استفاده می‌کنند و باید طراحی خود را با spec های تعیین شده ارزیابی کنیم تا نتیجه قابل قبول باشد.

performance: به صورت کلی کارایی را نشان می‌دهد که می‌توان دیدگاه‌های مختلف مثل پرعت و ... را در نظر گرفت.

throughput: حالت خاصی از performance است که کارایی را براساس سرعت اجرای یک برنامه در CPU نشان می‌دهد.

$$\text{performance} \propto \frac{1}{\text{execution time}}$$

مثال: دو کامپیوتر A و B موجود است، برنامه X روی کامپیوترهای A و B به ترتیب 10 و 15 ثانیه طول می‌کشد. کامپیوتر A چند برابر سریعتر از کامپیوتر B است؟

$$P_A \simeq \frac{1}{t_A} \qquad \frac{P_A}{P_B} = \frac{t_B}{t_A} = \frac{15}{10} = 1.5$$

$$P_B \simeq \frac{1}{t_B}$$

زمان اجرا

execution time = تعداد کلاک‌های حافظه × $\frac{1}{f_A}$ ← فرکانس پردازنده‌ای

X

تعداد دستورات برنامه

$$= \sum_{i=1}^{n} \text{زمان اجرای دستور } i = \sum_{i=1}^{n} \text{تعداد کلاک‌های دستور } i \times \frac{1}{f_A} = \sum_{i=1}^{n} CPI_i \times \frac{1}{f_A}$$

Clock Per Instruction

$$= \frac{I_C \times CPI}{f}$$

PqPCO

* تمرین ۵: MIPS پردازنده خود را حساب کنید

Subject:
Year. Month. Date. ( )

CPI : clock per Instruction

برای بدست آوردن CPI بصورت کلی برای هر پردازنده یا هر برنامه از میانگین CPI های دستورات استفاده می‌کنیم

IPC : Instruction per clock

در پردازنده‌های جدید با تکنیک‌های super scalar ممکن است با هر کلاک چند دستور وارد شود که از طریق این پارامتر می‌توان تعداد دستورها در هر clock را بدست آورد.

MIPS : Million Instruction per Second

$$MIPS = \frac{\text{تعداد دستورات اجرا شده}}{\text{مدت زمان طول کشیده برای دستورات (s)}} \times \frac{1}{10^6}$$

چند میلیون دستور در یک ثانیه اجرا می‌شود.

$$MIPS = \frac{I_c}{\text{exec. time}} \times 10^{-6} = \frac{f}{CPI} \times 10^{-6}$$

نکته: MIPS از CPI مهم‌تر است زیرا با افزایش CPI تعداد فرکانس و تعداد کلاک‌ها هم افزایش می‌یابد و این بیان‌گر بالا رفتن سرعت پردازنده نیست و باید بدانیم تعداد دستورات اجرای در مدت زمانی که clock می‌زند چقدر است و نتیجه MIPS می‌تواند بیان‌گر performance باشد.

پردازنده‌های برنامه MIPS هم وجود دارد که یک پردازنده معمولی و غیر صفحه‌ای با کارایی خوب است.

تذکر: می‌توان به جای حساب کردن CPI به ازای هر دستور، دستوراتی که CPI نزدیک به هم دارند را در کلاس‌های جدا قرار دهیم.

$$\text{execution time} = \sum_{j=1}^{m} C_j \times CPI_j \times \frac{1}{f}$$

↓ تعداد دستورات در برنامه که از لحاظ CPI با یکدیگر مساوی هستند

PAPCO

Subject:

Year. Month. Date. ( )

برای افزایش performance در پردازنده ها ابتدا فرکانس را زیاد کردند تا زمان اجرای برنامه کاهش دهند ولی این افزایش تأثیر عکس بر روی cpi داشت و هر دستور مقدار کلاک بیشتری را مصرف می کرد. براساس این تفکر دو دسته پردازنده به وجود آمدند:

1) RISC (Reduced Instruction Set computer): ① دستورات ساده تری را در نظر گرفته و در عوض تعداد کلاک های هر دستور کاهش یافته. cpi در RISC کم تر از CISC است.
② محدود بودن تعداد کل دستورالعمل ها ③ رشته های آدرس دهی کم ④ ثبات های کم تری مصرف کرده است ⑤ تعداد دستورات در هر برنامه زیاد است. ⑥ تعداد operand کم ⑦ طول دستورات ثابت است.

2) CISC (Complex Instruction Set computer): ① دستورات پیچیده تر شده ولی زمان تعداد clock اجرای هر دستور افزایش می یابد ② هر دستور clock متفاوتی نیاز دارد ③ رشته های متنوع و زیاد بودن آدرس دهی ④ تعداد زیاد رجیسترهای داخل پردازنده ⑤ تعداد دستورات در هر برنامه کم ⑥ تعداد operand ها زیادتر ⑦ طول دستورات متغیر ⑧ زیاد بودن دفعات دستورالعمل ها

از هر دو نوع پردازنده ها استفاده می شود از CISC برای server ها و mainframe ها که نیاز به انجام پردازش پیچیده تر است ولی در کامپیوترهای معمولی که از اپلیکیشن CISC استفاده نمی شود از RISC استفاده می کنیم.

در پردازنده های Intel از هر دو ساختار باهم استفاده می شود که هسته آن RISC است و پردازش بیرونی پردازنده CISC است. در واقع بنظر می رسد دستورات پیچیده ای را اجرایی کند ولی در درون خود از دستورات ساده استفاده می کند.

PqPCO

* تمرین 6: قانون Ahmdal را ثابت کنید: - چنانچه f درصد از برنامه تنها بصورت سریال اجرا شود و الباقی p برابر تسریع کنید در آن صورت میزان افزایش سرعت

Subject:

Year. Month. Date. ( )

$$speed\ up = \frac{1}{f + \frac{(1-f)}{p}}$$

محاسبات کامپیوتری:

از جمله عملیات ریاضی + ، − ، * ، ÷ ، ... است که مهم ترین آنها جمع است زیرا به کمک آن می توان سایر عملیات را نیز پیاده سازی کرد.

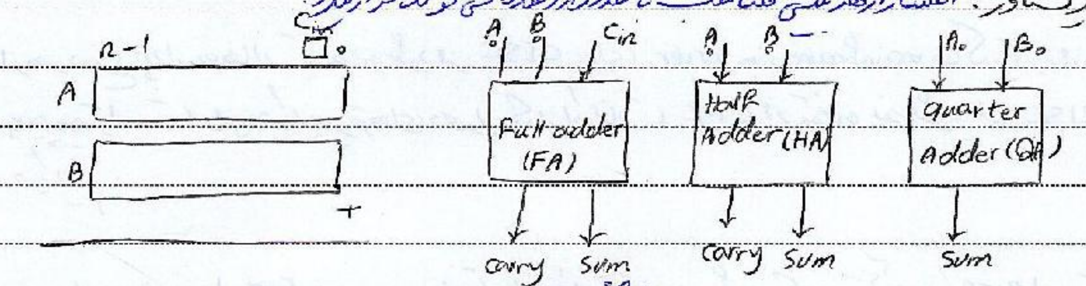
برای طراحی مدار جمع ابتدا باید مشخص کنیم که عمل جمع روی چه نوع از اعداد پیاده سازی می شود. اعداد را به صورت زیر تقسیم بندی می کنیم:

- اعداد
  - بدون علامت
  - علامت دار
    - اندازه و علامت
    - 2's complement

- اعداد
  - صحیح
  - اعشار
    - ممیز ثابت (Fixed point)
    - ممیز شناور (Floating point)

ممیز ثابت: محل اعشار مشخص است و جمع آن ساده است ولی اگر بخواهیم یک عدد صحیح بیشتر از تعداد بیت ها را نمایش دهیم قابل نمایش نیست حتی اگر قسمت اعشار خالی باشد.

ممیز شناور: اعشار از هر سمتی متناسب با عدد داده شده می تواند قرار بگیرد.



$$sum = A_0 \oplus B_0 \oplus C_{in}$$
$$carry = A_0 B_0 + B_0 C_{in} + C_{in} A_0$$

A_0	B_0	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$S_{um} = A_0 \oplus B_0$ $carry = A_0 . B_0$

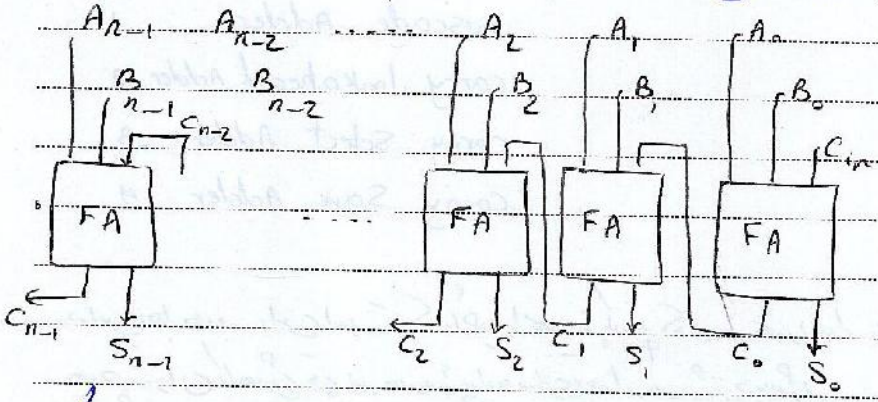
C_{in}	A_0	B_0	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

Subject:

Year. Month. Date. ( )

: Cascade Adder (1

مدار برای جمع دو عدد n بیتی به صورت زیر است:



به این نوع از مدار n-bit Ripple Adder یا n-bit Cascade Adder می گویند

حال برای انتخاب بهتر بودن مدار باید بتوان معیاری داشت که آن را مقایسه کرد.

$$\text{HW cost} = n \cdot \text{cost(FA)} = \begin{matrix} n & \text{xor} \\ n & \text{or} \\ 3n & \text{and} \end{matrix}$$

می توان به جای بیان گیت ها به صورت دقیق تعداد را براساس گیت به صورت کلی یا حتی تعداد FA ها بیان کنیم. گیت ها از ترانزیستور ساخته می شوند می توان با طراحی مدار به کمک ROM از گیت استفاده نکنیم زیرا ROM با دیود ساخته می شود.

میزان تأخیر مدار یعنی از لحظه ای که ورودی ها تعیین شده تا زمانی که خروجی آماده شود از رابطه زیر بدست می آید:

تأخیر گیت: d $\qquad S = \oplus \oplus \qquad : d$

تأخیر هر FA: Δ $\qquad C = + + \qquad : 2d$

زمان بیشترین Sum برای یکی از FA های میانی به اضافه زمان d برای بدست آوردن Sum خود.

$$\text{تأخیر delay} = \begin{cases} \text{تأخیر سیم} = (n-1)\Delta + d = (2n-1)d \\ \text{تأخیر } Cout = n\Delta = 2nd \end{cases}$$

Subject:
Year. Month. Date. ( )

انواع Adder ها عبارتند از :

1- cascade Adder

2- carry lookahead Adder

3- carry Select Adder

4- Carry Save Adder

در این ها هدف این است که تأخیر را کمتر کنیم که روند کمبود مدار Adder به صورت بالا است ولی هرچه زمان کاهش می یابد هزینه پیاده سازی مدار بیشتر می شود.

2- Carry Lookahead Adder :

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

$$C_1 = A_1B_1 + A_1C_0 + B_1C_0 = A_1B_1 + C_0(A_1+B_1) = A_1B_1 + (A_0B_0 + (A_0+B_0)C_{in})(A_1+B_1)$$

$$= A_1B_1 + A_0B_0(A_1+B_1) + (A_0+B_0)(A_1+B_1)C_{in}$$

با استفاده از روابط بالا می خواهیم sequential بودن عملیات جمع را از بین ببریم و مقدار C ها مستقل از یکدیگر بدست می آیند.

$$P_i = A_i \cdot B_i \qquad G_i = A_i + B_i$$

$\Rightarrow$

$$C_0 = P_0 + G_0C_{in}$$

$$C_1 = P_1 + P_0G_1 + G_0G_1C_{in}$$

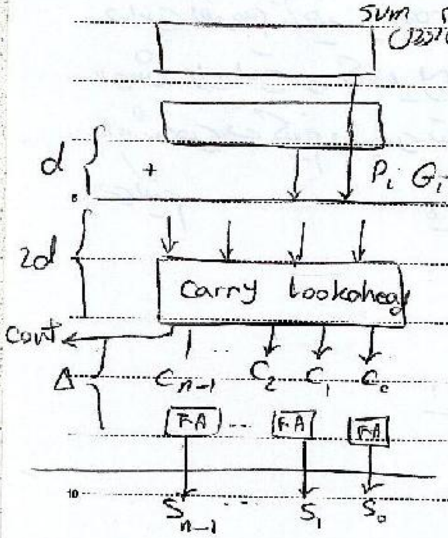
$$C_2 = P_2 + G_2(\quad) = P_2 + P_1G_2 + P_0G_1G_2 + G_0G_1G_2C_{in}$$

از آن جایی که P_i و G_i مستقل از یکدیگرند می توان دید که تأخیر مقدار همه را حساب کرد.

P4PCO

Subject:
Year. Month. Date. ( )

مطابق با شکل زیر همه P_i و G_i ها را به carry lookahead می‌دهیم تا موازی با هم C_i را تولید کنند

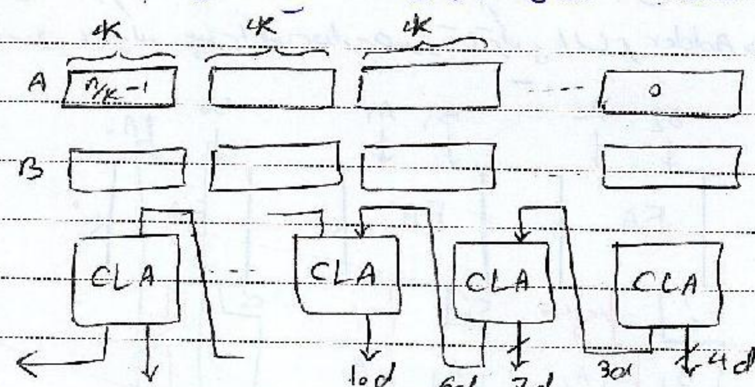


carry از مرحله قبل آماده شده همزمان با بیت آخر است.

Sum تأخیر $= 3d + d = 4d$

Cout تأخیر $= 3d = 3d$

نکته: با وجود اینکه این مدار یک مدار ترکیبی است (flip flop و feed back ندارد) ولی اگر بخواهیم n بیتی با sop یا pos تشکیل دهیم تعداد ورودی‌های گیت‌های and و or زیاد شده و چنین چیزی در عمل قابل پیاده‌سازی نیست. عملاً FA ها را با 4 ورودی طراحی می‌کنند که این شکل قابل حل باشد.



در واقع از ترکیب lookahead و cascade A استفاده می‌کنیم و اعداد را دسته‌های k تایی تشکیل می‌دهیم. اعداد داخل هر دسته را از طریق lookahead جمع می‌کنیم ولی برای جمع کردن k بیتی‌ها با یکدیگر از cascade A استفاده می‌کنیم.

Cout تأخیر $= 3(n/k)d$

Sum تأخیر $= 3(n/k)d + d$

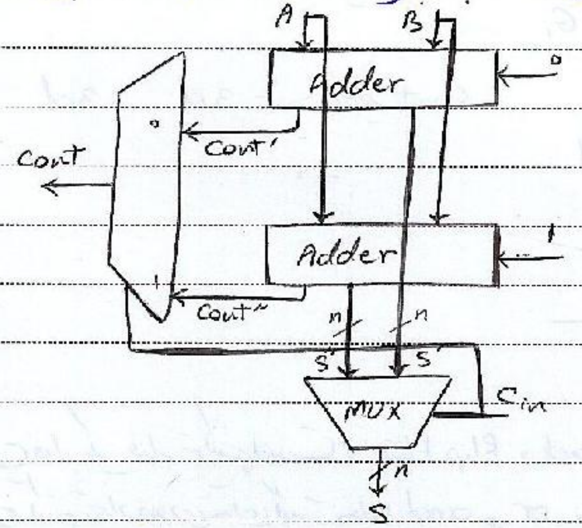
* تمرین ۷: مدار هریک از انواع جمع‌کننده را برای ۴ بیتی با استفاده از verilog توصیف کنید.

Subject.

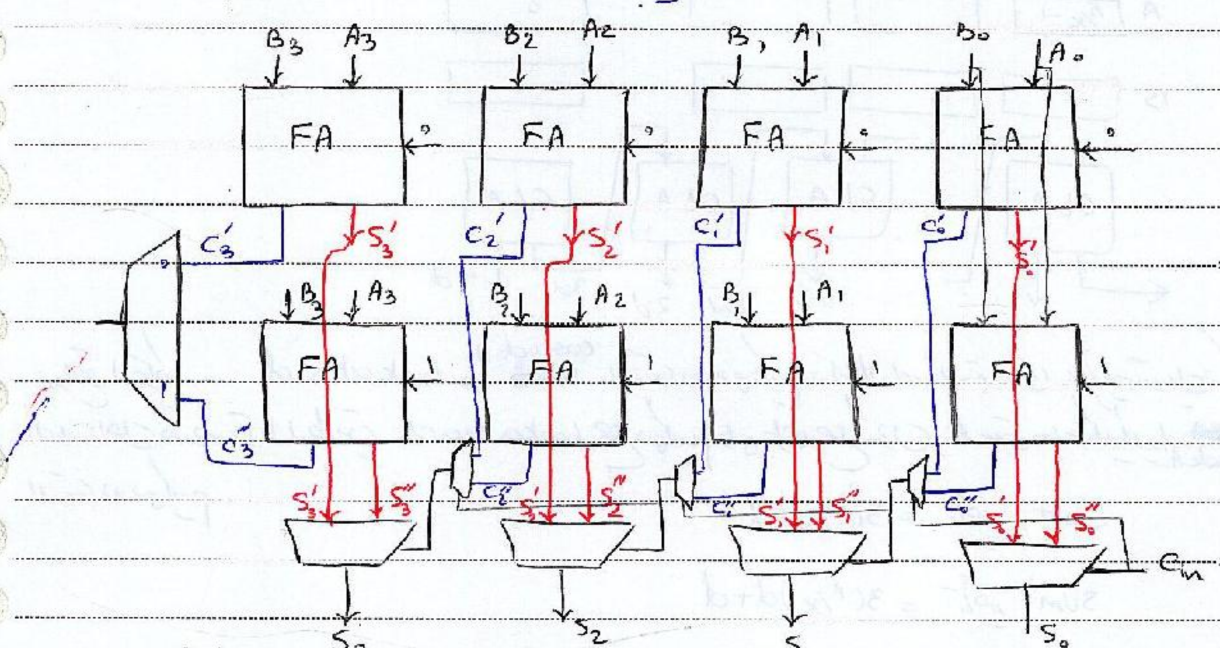
Year. Month. Date. ( )

13) Carry Select Adder : (CSA)

در این الگوریتم تأخیر carry را کاهش می‌دهیم زیرا بیشترین زمانی که برای محاسبه جمع در مدار صرف می‌شد برای این بود که باید carry از مرحله قبل آماده می‌شد. در این جا با carry صفر و یک مقدار جمع را مشخص می‌کنیم و سپس متناسب با carry درست از mux برای انتخاب نتیجه درست استفاده می‌کنیم

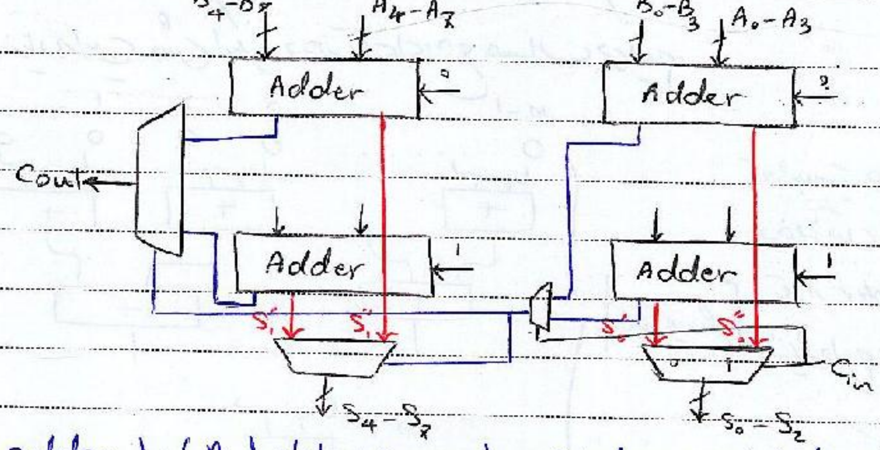


این مدار به خاطر اضافه شدن دو mux هم بیشتر زمان بیشتری نسبت به حالت‌های قبل مصرف می‌کند پس از این مدار به صورت ترکیبی با سایر Adder ها استفاده می‌کنیم



از مدار بالا برای جمع دو ۴ بیتی استفاده می‌کنیم و در هر دو حالت ورودی Cin صفر و یک حالات حساب می‌شود

Subject:
Year. Month. Date. ( )



$Delay(adder) + (\frac{n}{k})\, delay(mux) = (2kd) + 2(\frac{n}{k})d = 2nd$

ــــ k بیت k بیت جدا کرده باشیم.

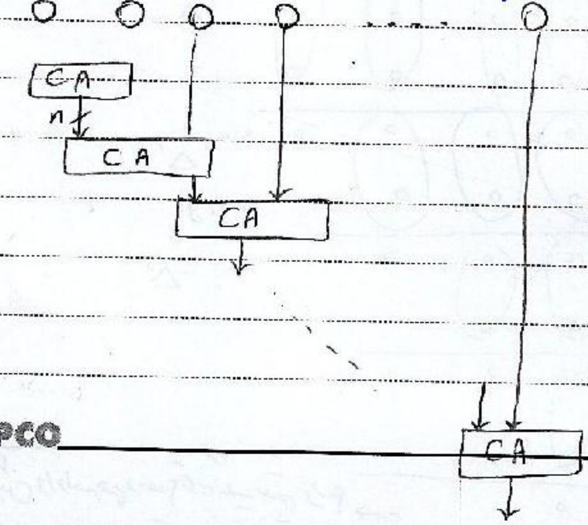
• $2kd + 2(\frac{n}{k})d < 2nd \implies k + \frac{n}{k} < n$ که اینجا بهترین k را در آوردن

اینجا مشخص می‌شود که اگر Adder ها را k بیتی در نظر بگیریم از CLA زمان کمتری مصرف می‌کند.

از این روش برای جمع k بیتی ها با یکدیگر استفاده می‌کنیم این مدار از نوع uniform است یعنی با وجود اینکه مقادیر جمع k تایی به صورت موازی تولید می‌شوند برای انتخاب جواب نهایی یک ارتباط بین carry ها وجود دارد و cascade بین carry ها برقرار است. پس می‌توان به جای محاسبه زودتر مقادیر زمان با صرف جمع کردن تعداد بیشتری بیت کنیم که به این مدل non uniform می‌گوییم.

4) Carry Save Adder:

می‌خواهیم m عدد n تایی را با یکدیگر جمع کنیم که در کامپیوترهای cisc کاربرد دارد. در واقع می‌خواهیم مداری برای جمع m عدد داشته باشیم با سرعت بالا طراحی کنیم.



روش اول:

ابتدا دو عدد را به وسیله CA ها جمع می‌کنیم سپس نتیجه را با عدد دیگر جمع می‌کنیم

هر چه به سمت پایین می‌رویم تأخیر بیشتر می‌شود چون تعداد بیت ورودی CA بیشتر می‌شود.

$Delay = (m-1)\, Delay(Adder)$

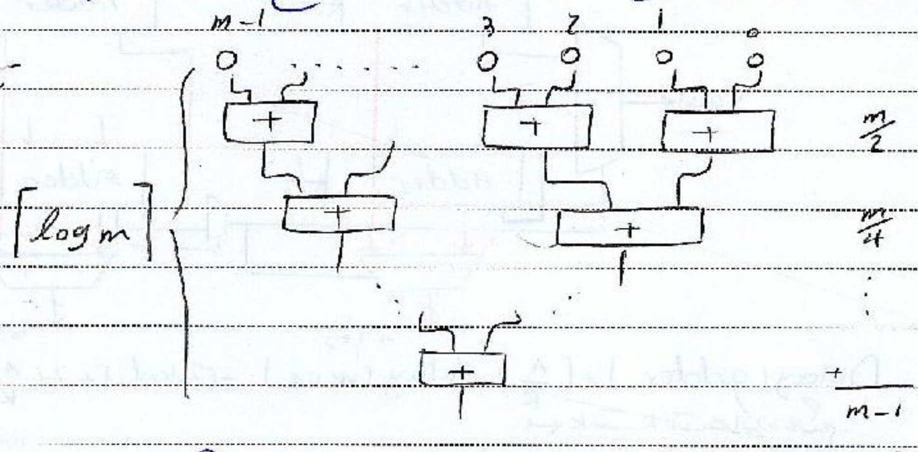
در هر مرحله یک بیت carry اضافه می‌شود در نتیجه

در نهایت تعداد بیت های خروجی برابر 2n است.

PAPCO

روش دوم:

به دلیل تأخیر زیاد در این روش از یک درخت برای جمع حساب می‌کنیم:

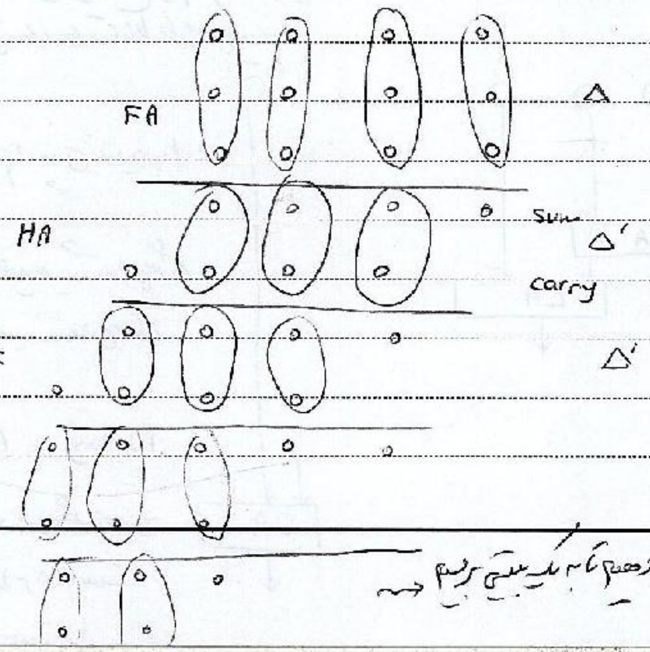


تو واریست هر مرحله کمتر از مرحله قبل است ولی تعداد Adder بیشتری سازماندهی

از آن جایی که در روش قبل تأخیر carry خیلی زیاد هست از روش carry save Adder استفاده می‌کنیم که در آن ابتدا بدون در نظر گرفتن carry اعداد را هم جمع می‌کنیم و carry ها را جدا ذخیره می‌کنیم سپس نتیجه جمع اعداد را با یک شیفت به سمت چپ از carry ها با یکدیگر جمع می‌کنیم

```
  9 1 2 4     ⟶ carry : 1 1 0 1
  3 6 5 2
  7 7 1 9 +
 ---------
  9 4 8 5     ⟹    9 4 8 5
                  1 1 0 1
                 ---------
                 2 0 4 9 5  ⟶ جواب نهایی
```

3 عدد 4 بیتی



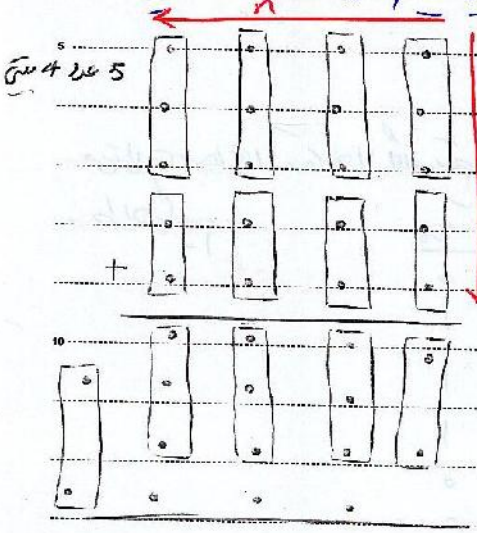
همین روش را ادامه می‌دهیم تا به یک بیتی برسیم

Subject:

Year. Month. Date. ( )

HW cost = 4 FA + 12 HA

Delay = $\Delta + 6\Delta' \simeq 7\Delta \simeq 14d$



اعداد
- مثبت
- علامت‌دار
  - اندازه و علامت: $A_s \; A_{n-1} \; A_0$
  - 2's complement: $A_{n-1} \; A_0$

A 1 0 0 0 0 0 0 1 (-127)

B + 1 0 0 0 0 0 0 1 (-127)

F 1 | 0 0 0 0 0 0 1 0 (-254)

$V = A_{n-1} B_{n-1} \overline{F}_{n-1} + \overline{A}_{n-1} \overline{B}_{n-1} F_{n-1}$

$V = C_{n-2} \oplus C_{n-1}$

Subject:

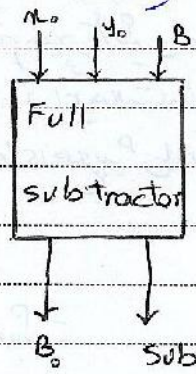
Year. Month. Date. ( )

تفریق کننده ها:

برای تفریق دو عدد 2's complement عدد دوم را نیست کرده و با عدد اول جمع می‌کنیم.

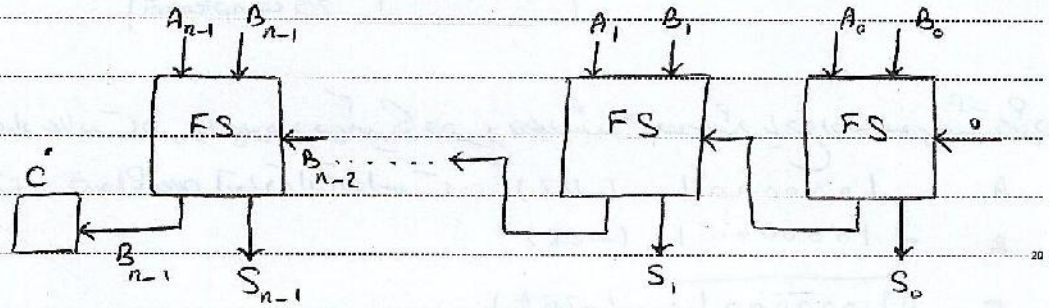
```
  1 1 1 1 1 1 1 0
- 0 0 0 0 0 0 0 1
-----------------
  1 1 1 1 1 1 1 0      254
```

می‌توان به جای استفاده از الگوریتم جمع و تفریق کننده ها مستقیماً یک تفریق کننده (Full Subtractor) طراحی کنیم.



x_0	y_0	B_{-1}	Sub	B_0
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

تفریق دو عدد n بیتی به صورت زیر است:



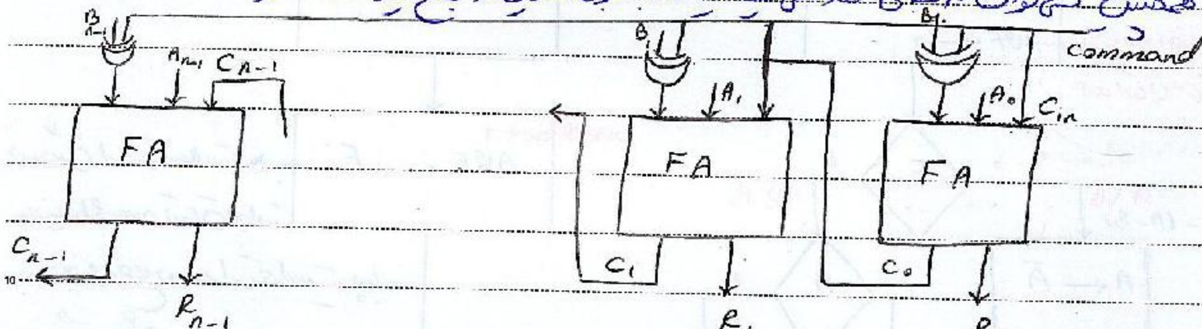
$$Sub = \bar{x}_0 (y_0 \oplus B_{-1}) + x_0 (y_0 \odot B_{-1})$$

$$B_0 = \bar{x}_0 (y_0 + B_{-1}) + x_0 y_0 B_{-1}$$

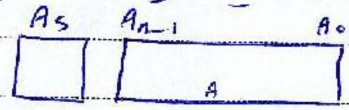
?? HW cost : nFS=

Delay
$$\begin{cases} Borrow = n\delta \\ Sub = n\delta \end{cases}$$

از روش زیر می‌توان برای تفریق اعداد بدون علامت و اعداد علامت دار از نوع 2's complement استفاده کرد.
همچنین می‌توان به جای طراحی یک گیت جدا برای تفریق از جمع نیز استفاده کرد:



یکی دیگر از انواع اعداد، اندازه و علامت است که می‌توان مطابق جدول زیر عملیات آن را محاسبه کرد



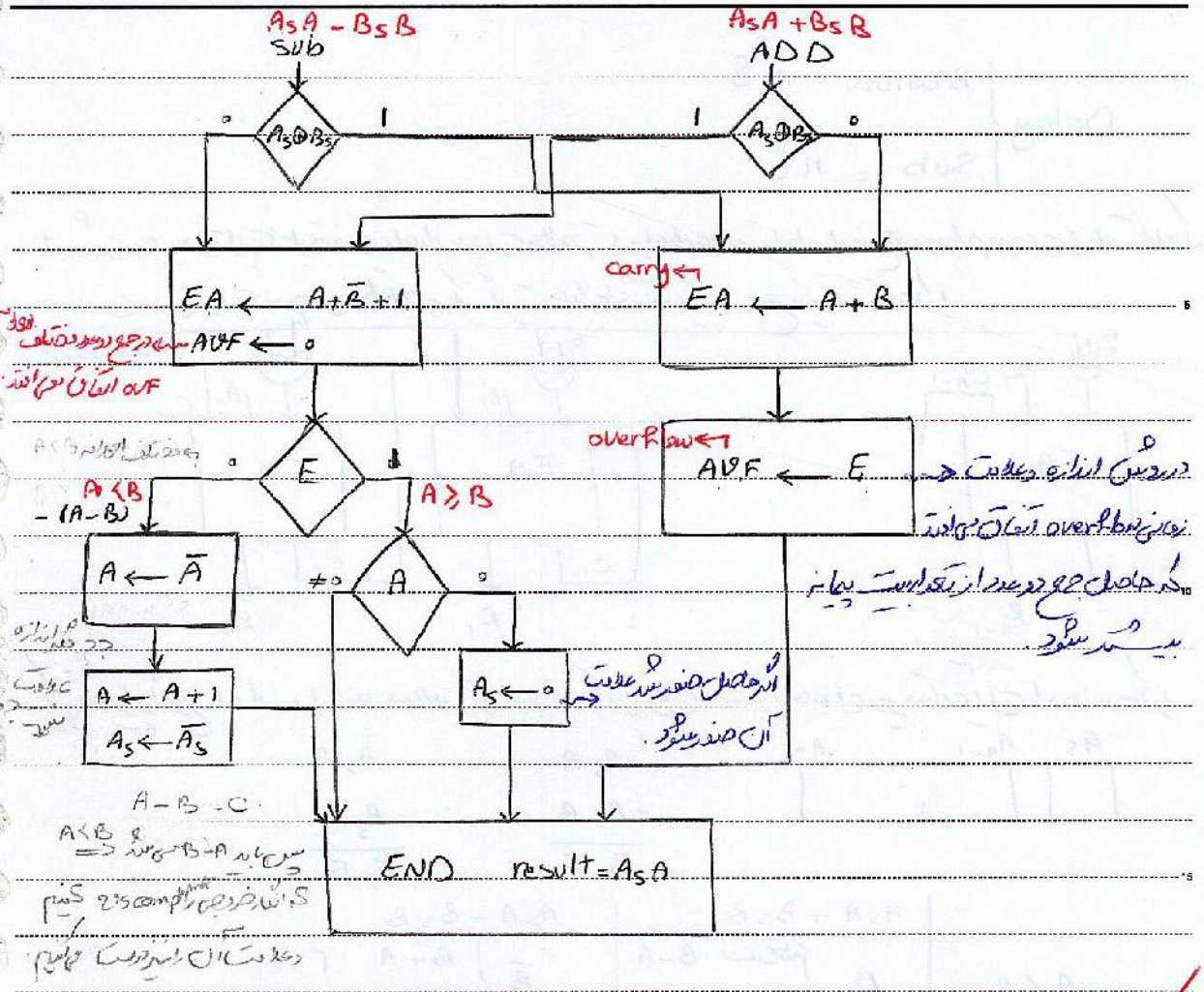
$$\begin{array}{r} A_S\ A \\ -\ B_S\ B \\ \hline F_S\ F \end{array} \qquad \begin{array}{r} A_S\ A \\ -\ B_S\ B \\ \hline F_S\ F \end{array}$$

	$A_sA + B_sB$	$A_sA - B_sB$
$A < B$ اندازه	B_S { مختلف العلامه $B - A$	$\overline{B_S}$ { $B + A$ ۲
	هم علامت $B + A$	$B - A$ ۵
$A = B$	0 { مختلف علامت 0	0 { 0 ۵
	A_S { هم علامت $A + B$	A_S { $A + B$ ۲
$A > B$	A_S { $A - B$ "	A_S { $A + B$ ۲
	$A + B$ "	$A - B$ ۵

طراحی صفحه‌ی بعد جدول بالا را پیاده‌سازی می‌کند و امکان محاسبه جمع و تفریق را به طور همزمان می‌دهد.
زمانی که دو عدد مثبت را از یکدیگر کم می‌کنیم از طریق مقدار carry می‌توان عدد بزرگتر را مشخص کرد.

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

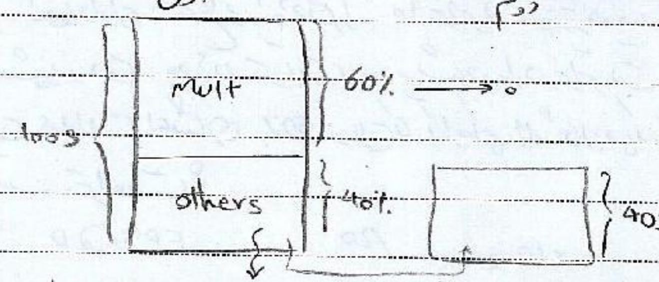
حال تفریق را از طریق 2's complement انجام داده و تنظیم علامت تفریق معمولی می‌شود.



این فلوچارت را براساس طراحی سخت افزاری می‌توان به دو صورت دنبال و آنکردن پیاده سازی کرد. در صورتی که هدف ما نباشد از جمع کننده‌های جدا استفاده کنیم نیازی به clock نداریم ولی در صورتی که از حاصل جمع قبلی استفاده می‌کنیم باید clock اضافه کنیم و دنبال کردن می‌شود.

Subject:
Year. Month. Date. ( )

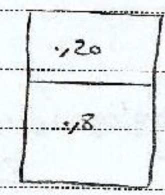
مثال: یک برنامه کامپیوتری در ۱۰۰ ثانیه اجرا می‌شود که ۶۰ ثانیه مربوط به عمل های ضرب است دستورالعمل های ضرب چقدر سریعتر شود تا اجرای برنامه ۲,۵ برابر سریعتر گردد؟



در نتیجه سرعت باید به سمت بی نهایت برود

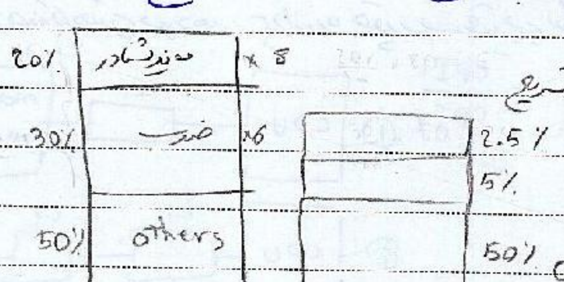
هیچ جایی برای ضرب باقی نمی‌ماند — دست نخورده باقی می‌ماند و حذف می‌شود

مثال: CPU دارم که شامل FA ای است که ۲۰٪ آن را تشکیل می‌دهد. میزان سرعت FA ده برابر بالا می‌رود سرعت کل چقدر بهبود می‌یابد



$$\text{speed up} = \frac{1}{0.8 + \frac{0.2}{10}} = \frac{1}{0.8 + 0.02} = \frac{1}{0.82} = \frac{100}{82}$$

مثال: یک برنامه با زمان اجرای 80s به صورت متوالی بر روی یک پردازنده انجام می‌شود از این زمان 20٪ برای دستورات ممیز شناور، 30٪ برای دستورات ضرب اعداد صحیح است و باقی‌مانده برای سایر دستورات استفاده می‌شود. فرض کنید با ایجاد تغییراتی در این پردازنده دستورات ممیز شناور 8 برابر و دستورات ضرب اعداد صحیح 6 برابر سریع‌تر شوند. میزان تسریع برنامه چقدر است؟



$$\text{میزان تسریع} = \frac{100\%}{57.5\%}$$

$$\text{قانون آمدال} = \frac{1}{(1-0.3-0.2) + \frac{0.2}{8} + \frac{0.3}{6}} = \frac{1}{0.575}$$

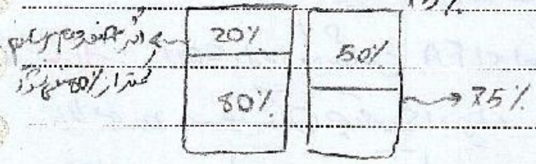
PAPCO

Subject:
Year. Month. Date. ( )

مثال: تابع ریشه دوم اعشاری یک برنامه گرافیکی به طور معمول به کار می رود. فرض کنید زمان صرف اجرای تابع ریشه دوم ۲۰٪ برنامه گرافیکی مورد نظر باشد دو راه کار برای تسریع این برنامه گرافیکی پیشنهاد می شود.
پیشنهاد اول: تابع ریشه دوم را ۱۰ برابر سریعتر کنید.
پیشنهاد دوم: همه دستورات ممیز شناور را دو برابر تسریع بخشید.
این عملیات اعشاری ۵۰٪ زمان کار گرافیکی را اشغال می کند کدام یک از دو پیشنهاد برای اجرای برنامه گرافیکی مناسب تر است؟

82% x10 ← %20 FPSQR

75% x2 ← %50 FP

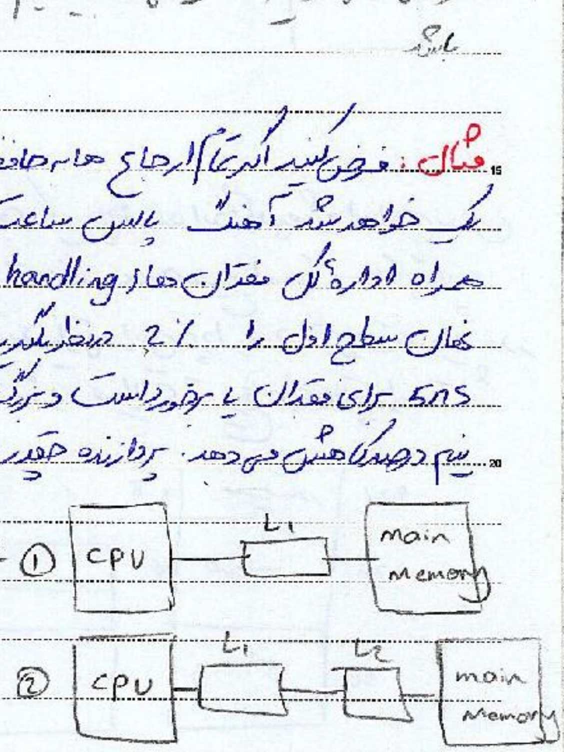


یعنی در هم خواهیم سیستمی را سریع کنیم باید بخش عمده را انتخاب می کنیم که درصد افزایش سرعت آن بیشتر باشد.

مثال: فرض کنید اگر تمام ارجاع ها به حافظه نهان سطح اول برخورد بنمایند CPI پردازنده برابر یک خواهد شد آهنگ پالس ساعت برابر 5GHz است. زمان دسترسی به حافظه اصلی را همراه اداره کل فقدان ها (miss handling) 100ns و آهنگ فقدان (miss rate) درصد دستورالعمل در حافظه نهان سطح اول را ۲٪ در نظر بگیرید در صورت افزودن یک حافظه ثانویه که دارای زمان دسترسی 5ns برای فقدان یا برخورد است و بزرگی آن به قدری است که آهنگ فقدان های حافظه اصلی را نیم درصد کاهش می دهد. پردازنده چقدر سریعتر می شود؟

$$\frac{Execution_1}{Execution_2} = \frac{\frac{I_C \times CPI_1}{R}}{\frac{I_C \times CPI_2}{R}} = \frac{CPI_1}{CPI_2}$$

پردازنده تغییر نکرده است.



hit rate = 98%

۰.۰۴ = ۰.۲ x ...

Subject:
Year. Month. Date. ( )

cache L1:

ایده‌آل CPI = 1 ⟹ از cache L1 خارج نمی‌شود

نرخ خطاها miss = 2%

$n \begin{cases} \text{Add 1} \\ \text{Add 1} \\ \vdots \end{cases}$

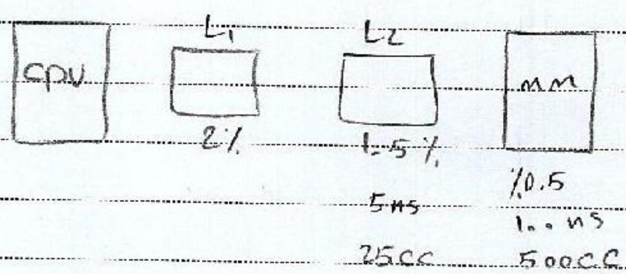
$\frac{100\,ns}{0.2\,ns} = 500$ ⟹ کلاک طول می‌کشد تا mm پاسخ دهد.

0.2 ns ⟵ میزان هر clock

$2 \begin{cases} \text{load } 1+m \\ \text{load } 1+m \end{cases}$

$CPI = 1 + 2\% \, (500) = 11$

cache L2:



$1 + 1.5\% \times 25 + \%0.5 \times 525$

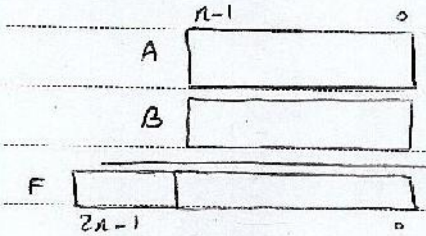
$1 + .2 \times 25 + \%0.5 \times 500 = 4$

54

Subject:
Year. Month. Date. ( )

ضرب کننده‌ها : Multiplier

در ضرب دو عدد n بیتی، یک عدد 2n بیتی بدست می‌آید زیرا در هر مرحله بدست آوردن partial product یک بیت به سمت چپ شیفت می‌دهد.

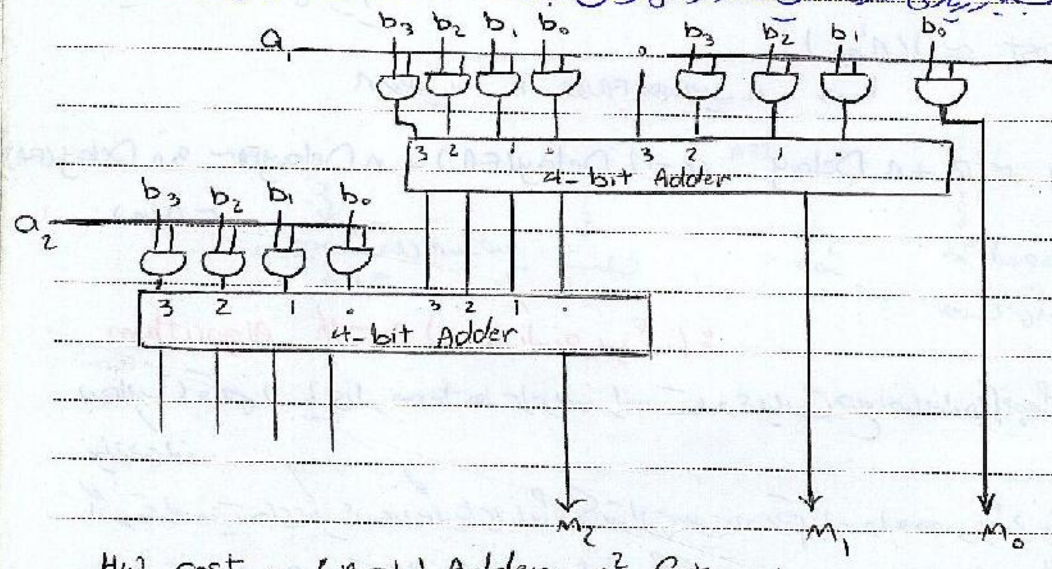


$$\begin{array}{r} 59 \\ \times\ 34 \\ \hline 236 \\ +\ 177\ \ \\ \hline \end{array}$$

236 : partial product (ضرب جزئی)

$$\begin{array}{r} a_{n-1} \cdots a_1 a_0 \\ \times\ b_{n-1} \cdots b_1 b_0 \\ \hline \cdots 0\ 0\ 0 \end{array} \quad \begin{cases} \text{if } b_0 = 0 : 0\,0\,0 \ldots \\ \text{if } b_0 \neq 0 : a_{n-1}\, a_{n-2} \ldots a_0 \end{cases}$$

یکی از روش‌های بدست آوردن حاصل ضرب این است که همه بیت‌های عدد دوم را در عدد اول ضرب کنیم که این delay بسیار زیادی دارد. این مدار را می‌توان به کمک Adder ها طراحی کرد



HW cost = (n-1) Adder + n^2 Gate$_{And}$

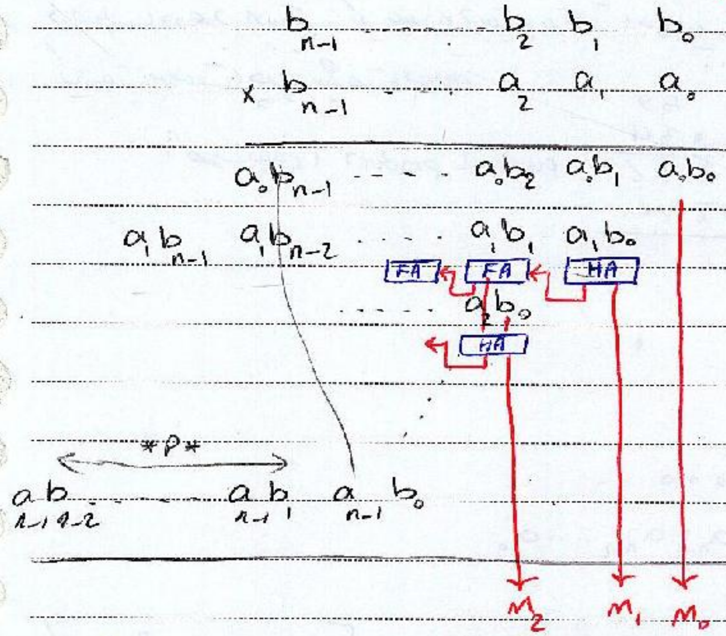
هر دو بیتی باهم and می‌شوند

Delay = (n-1) Delay$_{(Adder)}$ + d

تمام بیت‌ها که نیاز داریم gate$_{And}$ است در زمان d and ها قبل از آمدن خروجی Adder بدست آمده‌اند.

Subject:
Year. Month. Date. ( )

8 (ضرب کننده آرایه‌ای) Array Multiplier



برای انجام ضرب به این صورت یک cascade افقی و یک cascade عمودی برای جمع حاصل‌ضرب بیت‌ها در نظر می‌گیریم.

$HW\ cost \simeq O(n^2_{FA})$

$\longrightarrow$ n ردیف $\times$ تعداد FA در هر ردیف n

$Delay \simeq d + n\,Delay(FA) + (n-1)\,Delay(FA) + n\,Delay_{FA} \simeq 3n\,Delay(FA) \simeq O(n)$

- d: همه‌ی and ها مستقل از هم موازی درست می‌آیند.
- $n\,Delay(FA)$: سطر
- $(n-1)\,Delay(FA)$: ستون
- $n\,Delay_{FA}$: n بیت در حالت *P* را بیت‌ها جمع می‌کند.

Booth Algorithm (ضرب کننده بوث):

در حالت قبلی بسیاری از term ها برابر صفر است و عملیات جمع اضافه انجام می‌شود که تأخیر مدار را زیاد می‌کند.

اگر تعداد بیت‌های یک در عدد به گونه‌ای باشد که بتوان بین دو رقم 1- را محصور کرد نیازی به ضرب نداریم و با نوشتن عدد به صورت توانی از دو و از طریق شیفت این توان 2 ها انجام می‌شود.

طریقه‌ی محصور کردن یک‌های درون عدد با 1- و 1 به این صورت است که اولین یک از سمت راست را 1- قرار داده و order یک خانه بعد از آخرین یک‌های توالی را 1 قرار می‌دهیم و می‌توانیم گروه‌های

Subject.

Year. Month. Date. ( )

یک مقدار را به این گونه نمایش دهیم.

$+15 = 0000\,1111 = 2^4 - 2^0$

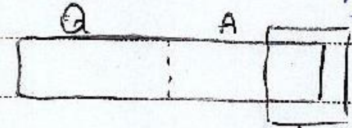
$-45 = 1101\,0011 = (2^2 - 2^0) + (2^5 - 2^4) + (-2^6)$

اگر به بیت صفر هم نرسد نیازی به نوشتن نداریم.

$0111\,0101 = (2^1 - 2^0) + (2^3 - 2^1) + (2^7 - 2^4)$

$1011\,1000 = (2^6 - 2^3) - 2^7$

در نتیجه برای پیاده سازی سخت افزاری آن به گونه ای عمل می کنیم که دو n بیتی Q و A را به عنوان نتیجه خروجی در نظر گرفته و در هر مرحله دو بیت سمت راست A را چک می کنیم و بر اساس وجود هر یک از الگوهای زیر عملیات مربوط به آن را انجام می دهیم.



00 : شیفت به سمت راست

11 : "

10 : $QA - B \rightarrow Q.A$

01 : $QA + B \rightarrow QA$

بدترین حالت برای عدد به این صورت است: ...01010101

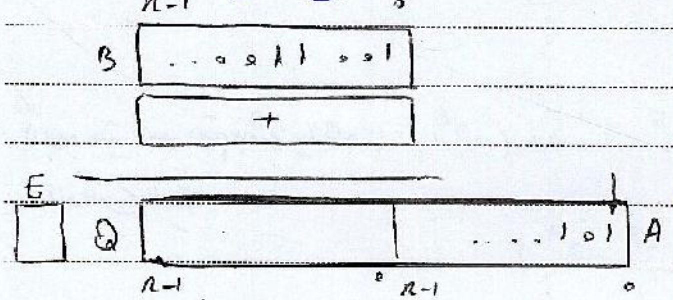
...10101010

که مجبور است در هر مرحله عمل جمع را انجام دهد.

نکته: در انجام ضرب کاغذی یا دودویی همین مکانیزم را با یک بیت انجام می دهیم به این صورت که Q و A را در انتها هم به عنوان خروجی در نظر می گیریم که در ابتدا باید Q صفر باشد. در ضرب معمولی به ازای هر ضرب میانی باید نتیجه را ابتدا به چپ شیفت دهیم و سپس با خروجی جمع کنیم به جای اینکه خروجی را

PAPCO

ابتدا یکی به راست شیفت می‌دهیم که در پس حاصل ضرب را با خروجی جمع می‌کنیم



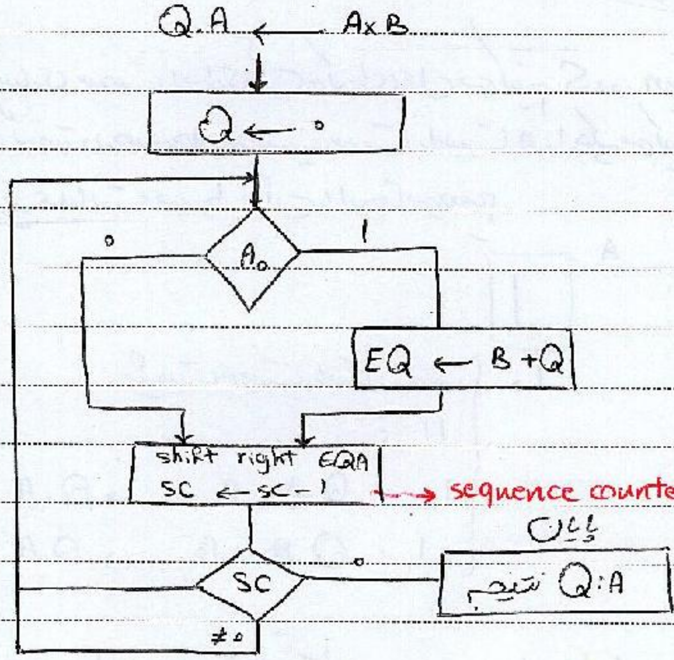
۱ـ به اندیس A_0 نگاه کن

۲ـ اگر $A_0 = 0$ ـ

اگر $A_0 = 1$ ← $B+Q \to Q$

۳ـ یک واحد به سمت راست شیفت کند

$B \times A = Q.A$



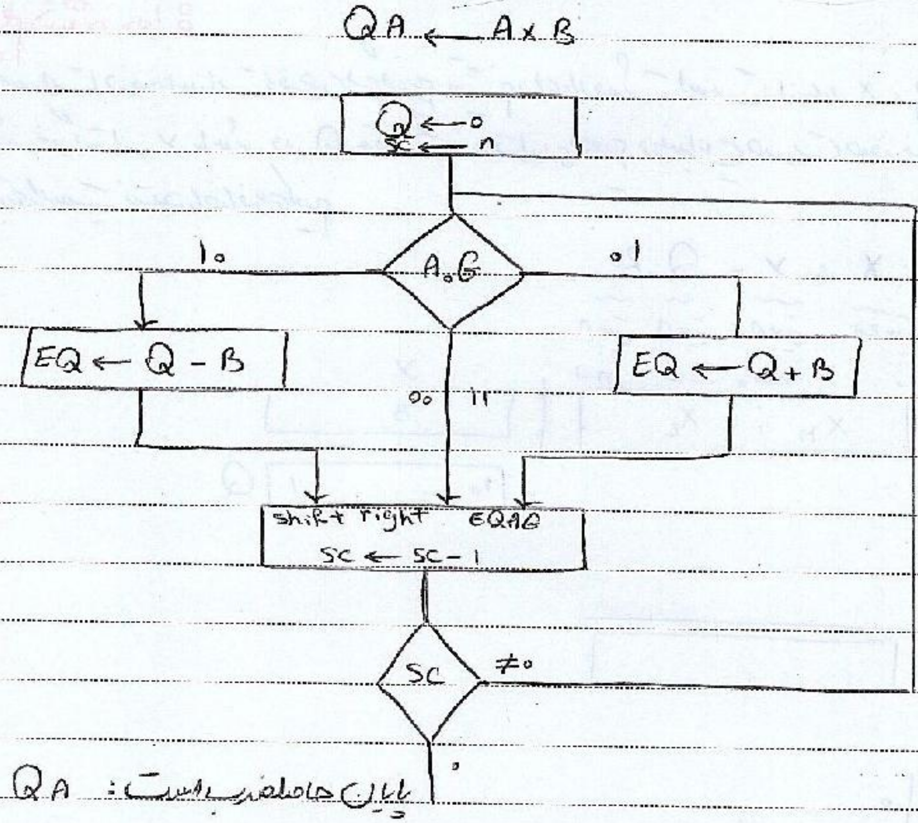
→ sequence counter عمل شیفت به طول n بار باید انجام شود.

الگوریتم بالا تاخیر بیشتری نسبت به ضرب آرایه‌ای دارد ولی هزینه سخت افزاری آن کمتر از آرایه‌ای است.

در الگوریتم booth علاوه بر وجود E به عنوان carry یک بیت در سمت راست آرایه A به نام G قرار می‌دهیم و هنگام مقایسه به مقادیر $A_0 G$ دقت می‌کنیم.

Subject:
Year. Month. Date. ( )

booth algorithm



به صورت متوسط الگوریتم بوث بهتر از الگوریتم قبلی کار می کند. برای انجام عمل تفریق به وسیله جمع کننده از حالتی استفاده می کنیم که بوسیله یک عدد مربوط به command جمع یا تفریق را مشخص می کنیم که command آن برابر است با ($A_0\overline{G}$).

الگوریتم بوث برای 2's complement و اعداد مثبت کار می کند.
الگوریتم ترتیبی و آرایه ای برای 2's complement کار نمی کنند در نتیجه اعداد را از 2's com. در می آوریم و مثبت می کنیم و عمل ضرب را انجام می دهیم و با توجه به علامت اعداد جواب را می سازیم.

تمرین اختیاری: ثابت کنید که شرط زیر برای مشخص n بیتی شدن Q لازم و کافی است:

Q بیش از n بیت است ← $X_H \geq Y$ می‌شود.

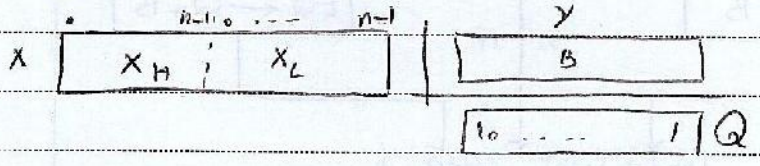
Subject:
Year. Month. Date. ( )

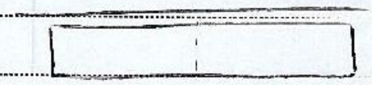
تقسیم کننده‌ها

برای محدودیت در محاسبات فرض می‌کنیم تقسیم در اعداد مثبت است. از عدد X، n بیت را انتخاب کرده که بزرگتر از Y باشد در Q مقدار یک را قرار می‌دهیم در غیر این صورت صفر می‌گذاریم. در هر مرحله یک بیت به سمت نتیجه اضافه می‌کنیم.

$$X \div Y = Q , R$$

2n بیتی — n بیتی — n بیتی — n بیتی





$$AVF = \begin{cases} 0 \\ 1 & \begin{cases} \frac{n}{0} & \text{تقسیم بر صفر} \\ \text{Q n بیتی نباشد} \end{cases} \end{cases}$$

اگر در اولین بار A_H بزرگتر از Y باشد Q بزرگتر می‌شود.

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

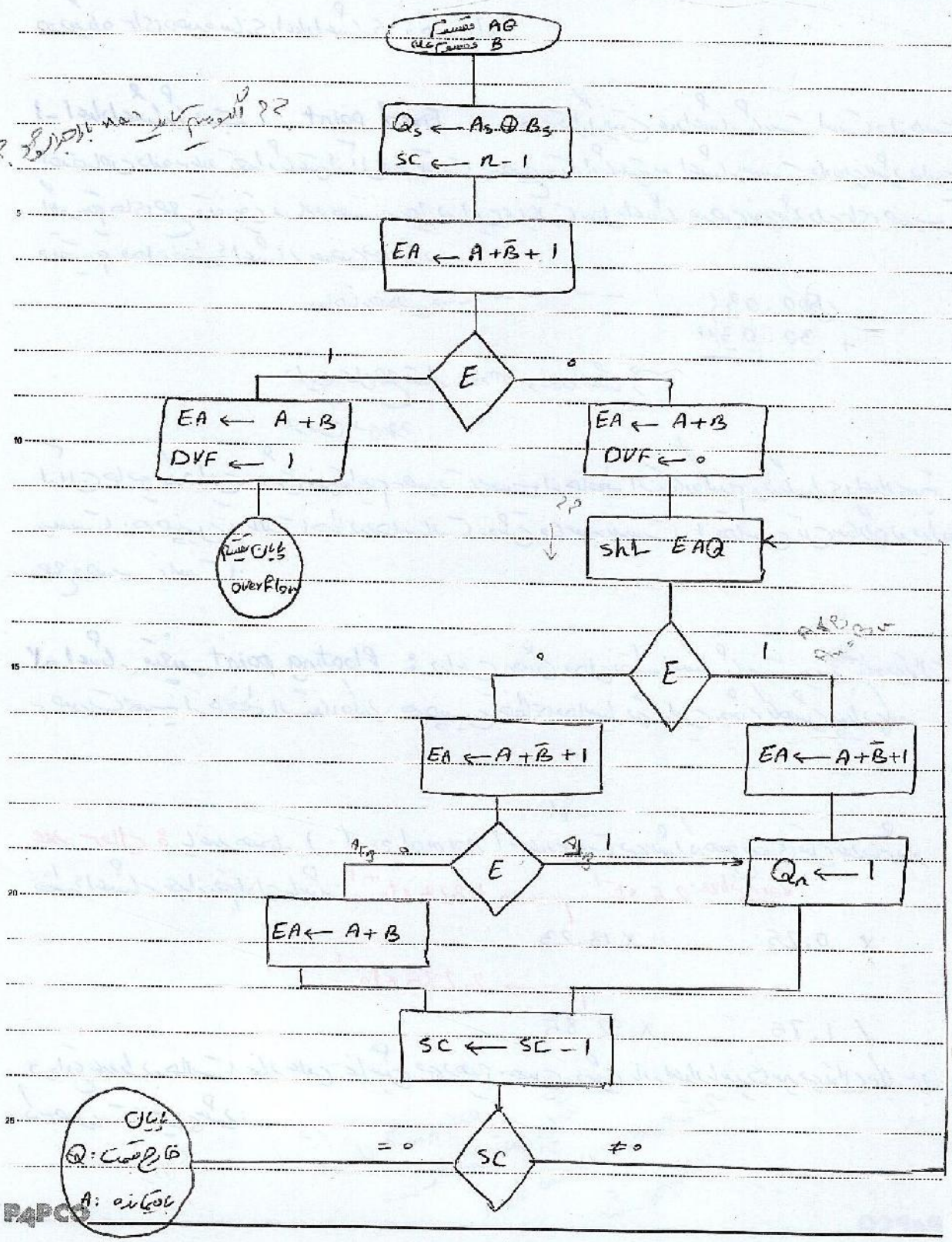
از طریق بیت carry می‌توان مشخص کرد که آیا OVF در حالت دوم اتفاق افتاده یا نه.

در علامت‌دار صورت 2n بیتی را A.Q در نظر گرفته و Y را با B نشان می‌دهیم. OVF نشان‌دهنده بیت مربوط به overflow است. در تقسیم دو عدد از نمایش اندازه و علامت استفاده شده است که بدون در نظر گرفتن علامت دو عدد را تقسیم کرده و سپس نتیجه نهایی را تعیین علامت می‌کنیم.

PAPCO

Subject:

Year. Month. Date. ( )



Subject:
Year. Month. Date. ( )

دو دیدگاه برای ذخیره سازی اعداد اعشاری وجود دارد:

۱- اعداد اعشار ثابت Fixed point : محل قرارگرفتن عدد اعشار ثابت است و از n بیت اختصاص داده شده تعداد ثابتی از آن به قسمت صحیح و تعداد ثابتی به اعشار نسبت داده می شود در این الگوریتم های جمع و تفریق ساده هستند زیرا در این عملیات محل اعشار عوض نمی شود ولی برای ضرب و تقسیم محل نقطه ی اعشار عوض می شود.

500.031
+ 30.034

--- . --- ساختار خروجی

در این عمل جمع مقدار msb و lsb اعشار از بین می رود.

از آن جایی که در این روش نمی توانیم به صورت بهینه از حافظه استفاده کنیم و نگهداری اعداد بهینه نیست. همچنین به علت خطا در محاسبات روش مناسبی نیست. از محاسن آن سادگی در فرآیندهای جمع و ضرب است.

۲- اعشار متغیر Floating point : در این روش محل نقطه ی اعشار ثابت نیست و می توان به صورت بهینه از حافظه استفاده کرد. همچنین خطای محاسباتی در این روش کاهش پیدا می کند.

عدد نرمال یا عدد هنجار (normalized) به عددی گفته می شود که هیچ صفری قبل آن نباشد و نقطه ی اعشار بعد از رقم اول باشد.

× 0.25 → normalize → 2.5×10^{-1}

× 13.23 → $1.323 \times 10^{+1}$

✓ 1.75

× 92.85 → $9.285 \times 10^{+1}$

در واقع عدد را به حالت نماد علمی نمایش می دهیم. همین روش را در اعداد باینری نیز می توان به کار برد که به صورت زیر می شود:

Subject: Mantis Fraction اعشار، مانتیس

Year. Month. Date. ( )

توان، نما، exponent

$\times$ 101.111 → 1.01111×2^{2}

$\times$ 110.111 → 1.10111×2^{2}

✓ 1.001

$\times$ 101.001 → 1.01001×2^{2}

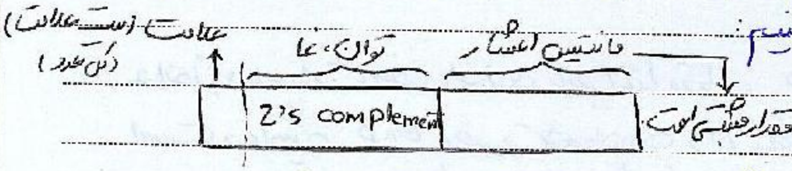
$\times$ 0.10101 → 1.0101×2^{-1}

$\times$ 0.11101 → 1.1101×2^{-1}

صفر را یک عدد هنجار شده در نظر گرفته و سایر اعداد حداقل یک 1 دارند که با شیفت دادن ها راست و چپ می توان هنجار کرد. در حالت کلی روش نرمال کردن اعداد به صورت زیر است: (ذخیره عدد)

۱- عدد را هنجار کنید.

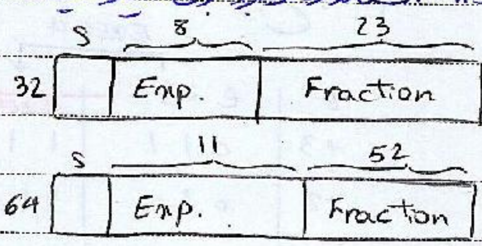
۲- نما و اعشار بدست آمده را در قالب ممیز شناور جایگزین کنید: در همه اعداد می دانیم که مقدار صحیح برابر 1 و بعد از آن نقطه اعشار وجود دارد پس نیازی به ذخیره کردن ندارد و تنها لازم است که مقدار اعشار و توان را ذخیره کنیم:



754: استاندارد موجود برای ذخیره یک عدد اعشاری در IEEE به صورت زیر است:

Single precision (دقت ساده)

Double precision (دقت مضاعف)



از ویژگی هایی که باید در نمایش به روش Floating point در نظر گرفته شود عبارتند از:

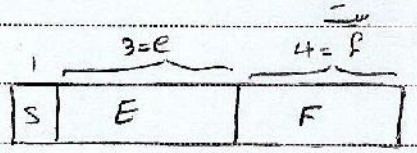
1) تعداد اعداد قابل نمایش: به علت محدود بودن تعداد بیت اختصاص داده به عدد تنها بازه ای از اعداد را می توان نمایش داد.

2) بازه اعداد قابل نمایش: محدوده اعداد قابل نمایش باید مشخص شود.

Subject.
Year. Month. Date. ( )

13 حداقل عدد مثبت و حداکثر عدد مثبت

14 روند محاسبات



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

برای نمایش عدد صفر به صورت مستقیم قابل انجام نیست ولی می‌توان حداقل عدد مثبت را صفر در نظر گرفت.

$(-1)^0 \times 1.0 \times 2^{E_{min}}$ ← صفر

قرارداد در ساخت مدار سخت افزاری باید صفر را به صورت دستی حساب کرد و دارد وارد اصلی مربوط به محاسبات نشود.

در انجام محاسبات سخت افزاری باید ابتدا مقدار Exp را مقایسه کنیم و مقدار آن از 4 تا F است و نمایش Exp به صورت 2's complement است. سپس می‌شود مدار مقایسه کننده exp ها پیچیده می‌شود برای برطرف کردن این مشکل یک مقدار offset را با عدد جمع می‌کنیم.

EXEC 4 ↓ +2^{e-1} bias1 → bias2

EXEC 3 →

E	e=3		$2^{e-1}-1$
+3	0 1 1	1 1 1	1 1 0
+2	0 1 0	1 1 0	1 0 1
+1	0 0 1	1 0 1	1 0 0
0	0 0 0	1 0 0	0 1 1
-1	1 1 1	0 1 1	0 1 0
-2	1 1 0	0 1 0	0 0 1
-3	1 0 1	0 0 1	0 0 0
-4	1 0 0	0 0 0	1 1 1 → نامعنی ها

$-2^{e-1} \leq E \leq 2^{e-1}-1$

Not a Number = NAN

Subject:

Year.　Month.　Date.　(　)

در واقع عددی که انتخاب می‌کنیم به عنوان offset یا همان emp جمع می‌شود برابر کوچکترین عدد منفی (4-) که ثبت شده است می‌باشد. این کار باعث می‌شود تا در نمایش عدد صفر همه بیت‌ها برابر صفر بشود یعنی هرگاه کل بیت‌های نمایش عدد برابر صفر باشد عدد را صفر در نظر می‌گیریم.
$(E_{min}=0)$

نکته: برای اینکه بتوانیم یک عدد خاص مثلاً حالت‌های مبهم یا اعداد NaN و ... را نمایش دهیم از $E_{mec}\ 2^{e}-1$ استفاده شد در نتیجه بازه انتهایی محدودتر شده و مقدار صفر یکی بالاتر رفته. الگوی 111 مربوط به نمایش اعداد خاص یا ناعددی است که از طریق مقادیر مختلف F الگوهای گوناگون خاص را نمایش می‌دهیم.

⚠ از آن جایی که اعداد نمایش داده شده در قالب اول Bias شده است برای مشخص عدد از فرمول زیر استفاده می‌کنیم:

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

| 1 | 001 | 1110 | → $= -1.1110 \times 2^{-3}$

$+0.25$ $\xrightarrow{\text{نرمال شود}}$ 1.1001×2^{-1} $\xrightarrow{\text{نمایش در بیت}}$ | 0 | 011 | 1001 |

0.11001 در مبنای 2　　$-1+4=3$

نکته: از آن جایی که استفاده از اعداد کوچکتر بیشتر است در نمایش اعداد کوچک دقت بیشتری داریم.

$0 \quad 1.\overbrace{00\ldots00}^{P} \times 2^{E_{min}}$

$0 \quad 1.00\ldots01 \times 2^{E_{min}}$

$\vdots$

$(1+\frac{1}{2^P}) \times 2^{E_{min}}$

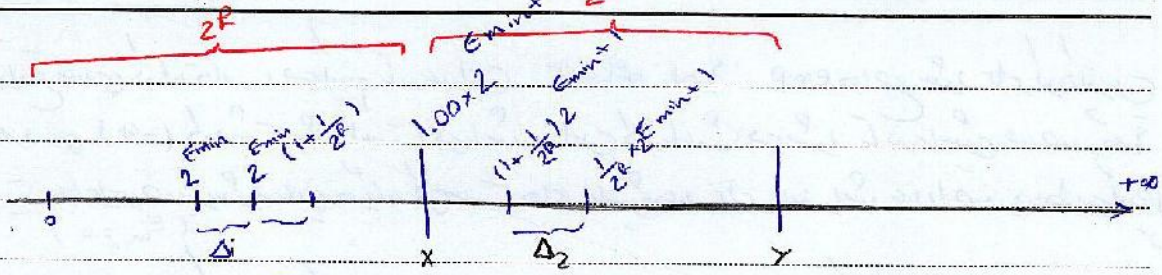
زیرا تعداد اعداد قابل نمایش در هر بازه با توان ثابت برابر است ولی فاصله این اعداد از هم بیشتر می‌شود.

PAPCO

Subject:

Year. Month. Date. ( )

* تعریف اپسیلون: $\begin{cases} 1+\frac{1}{2}+\frac{1}{3}+\dots+\frac{1}{n} \\ \frac{1}{n}+\frac{1}{n-1}+\dots+1 \end{cases}$



$\Delta = \frac{1}{2^P} \times 2^{E_{min}}$

اکتاو octave

همچنین بسمت بی‌نهایت می‌رویم بازه‌ها دوبرابر و Δها نیز دوبرابر می‌شوند ولی تعداد اعداد قابل نمایش در همهٔ آنها ثابت است. پس هرچه بسمت بی‌نهایت می‌رویم دقت کمتر می‌شود.

$\Delta_1 = 0.0\cdots 1 \times 2^{E_{min}} = 2^{-P}\cdot 2^{E_{min}}$ { کوچکترین عدد مثبت

$= 1.00\dots0 \times 2^{E_{min}}$

$1.00\dots1 \times 2^{E_{min}}$

$1.00\dots10 \times 2^{E_{min}}$

$\vdots$

$1.11\dots11 \times 2^{E_{min}}$

$\Delta_2 = 2^{-P}\cdot 2^{E_{min}+1} = 2\Delta_1$

{ $x = 1.00\dots0 \times 2^{E_{min}+1}$

$1.00\dots1 \times 2^{E_{min}+1}$

$\vdots$

$1.111\dots1 \times 2^{E_{min}+1}$

$y = 1.000\dots0 \times 2^{E_{min}+2}$

$N_{max} = +1.\underbrace{11\dots1}_{P} \times 2^{E_{max}} \Rightarrow N_{max} = (2-2^{-P})\times 2^{2^{e-1}-1}$

$N_{min} = -N_{max}$ بازهٔ اعداد قابل نمایش $= [N_{min}, N_{max}]$

* bias1 $E_{min} = -2^{e-1}$

* bias2 $E_{min} = -2^{e-1}+1$

PAPCO

Subject:
Year. Month. Date. ( )

برای بدست آوردن حداقل اعداد اعشار باید نوع bias عدد را استفاده [مشخص] کنیم که در اینجا اعداد را از نوع bias2 در نظر می‌گیریم

$$\begin{cases} N_{min.positive} = 1 \times 2^{-2^{e-1}+1} = +2^{-2^{e-1}+1} \\ N_{max.positive} = (2 - 2^{-f})\, 2^{2^{e-1}-1} \end{cases}$$

تعداد اعداد قابل نمایش $= 2 \times 2^{f} \times (2^{e} - 1)$

- 2 ← حالت های sign
- 2^{f} ← حالت های اعشار
- $(2^{e} - 1)$ ← تعداد حالت های توان به جز ناعددی

تعداد ناعددی های قابل نمایش $= 2 \times 2^{f}$

← توان تنها یک حالت دارد و با تعداد حالت های اعشار و علامت ناعددی ها بدست می‌آیند

عدد	علامت	نما	اعشار
عدد صفر	0	0	0
عدد مثبت	0	0	$\neq 0$
	0	$\neq 0$	$\neq 0$
	1	0	0
	1	0	$\neq 0$
	1	$\neq 0$	0
	1	$\neq 0$	$\neq 0$

⟹

	اعشار	نما	علامت
ناعددی	هر عدد	1111111	هر عدد
صفر	0	0	0
اعشاری (شرط اینکه)	هر عدد	0 → 1110 (به جز)	هر عدد

حالات مشترک آن زیاد است پس از جدول مقابل آن استفاده می‌کنیم

PAPCO

الگوریتم ADD دو عدد اعشاری ممیز شناور هنجار شده:

برای جمع دو عدد باید نماها را یکسان کنیم و سپس عملیات جمع را انجام دهیم. [نمای] کوچک را به نمای بزرگ تبدیل کنیم زیرا در بزرگ کردن نما(ی) شیفت به راست می‌دهیم و در صورت حذف شدن عدد در بیت، از کوچکترین و کم ارزش‌ترین بیت صرف نظر شده است.

ولی در صورتی که توان بزرگ را به کوچک تبدیل کنیم ممکن است بزرگترین بیت و با ارزش‌ترین از دست برود.

$A = (-1)^{S_A}\ 1.F_A \times 2^{E_A}$

$B = (-1)^{S_B}\ 1.F_B \times 2^{E_B}$

$$1.101 \times 2^{5} + 1.001 \times 2^{-3}$$

حالت اول / حالت دوم

$$\begin{array}{r} 1.101 \times 2^{5} \\ +\ 0.00000001001 \times 2^{5} \\ \hline 1.10100000001001 \times 2^{5} \end{array} \qquad \begin{array}{r} 110100000000 \times 2^{-3} \\ 1.001 \times 2^{-3} \\ \hline 110100000001.001 \end{array}$$

ارزش‌ها از دست می‌روند

مراحل ADD کردن به صورت زیر است:

1) مقایسه نماها

2) نمای عدد با نمای کوچک را به نمای عدد بزرگ می‌رسانیم (ردیف کردن نماها)

3) اعشار عدد کوچک را به تعداد اختلاف نماها به سمت راست شیفت می‌دهیم

4) عمل جمع معمولی روی اعشار انجام می‌دهیم — ضرب ← جمع ← overflow

5) عدد بدست آمده نرمال شود. — تقسیم ← تفریق ← underflow

چنانچه در محاسبات اعداد اعشاری ممیز شناور بعد از عمل محاسبات مقداری ناهنجار که با صفر شروع شده باشد برخورد در این صورت یک زیر ریز / جدول زیر underflow رخ می‌دهد

$$\begin{array}{r} 1.0101 \times 2^{2} \\ -\ 1.0001 \times 2^{2} \\ \hline 0.0100 \times 2^{2} \end{array}$$

در انجام عمل تفریق نیز همانند الگوریتم بالا عمل می‌کنیم با این تفاوت که شرط underflow را در نظر می‌گیریم.

?? اعداد اعشاری ?

Subject.
Year. Month. Date. ( )

الگوریتم ضرب در ممیز شناور:

۱) اعداد اعشار را در هم ضرب می‌کنیم.

۲) علامت نتیجه = $S_B \oplus S_A$

۳) نمای نتیجه = $E_A + E_B - bias = (E_A - b) + (E_B - b) + b$

۴) عدد بدست آمده نتیجه را نرمال کنید:

$$\begin{array}{r} 0.101 \times 2^2 \\ \times\ 0.111 \times 2^3 \\ \hline \end{array}$$

نکته: در این الگوریتم به ازای بالا بردن نماها با بایاس جمع می‌شوند در نتیجه هیچ وقت underflow اتفاق نمی‌افتد و تنها ممکن است overflow رخ دهد. که در نهایت آن را نرمال می‌کنیم

الگوریتم تقسیم در ممیز شناور:

۱) اعداد اعشار را به هم تقسیم می‌کنیم

۲) علامت نتیجه = $S_A \oplus S_B$

۳) نمای نتیجه = $E_A - E_B + b$

۴) عدد نتیجه را نرمال کنید.

در این الگوریتم نیز چون تقسیم از تفریق های متوالی ایجاد می‌شود تنها ممکن است underflow ~~over~~ flow به وجود آید.

PAPCO

Subject:
Year.　　Month.　　Date.　　(　)

تذکر: در انجام عملیات جمع و تفریق اعشار با ممیز ثابت الگوریتم ساده تر می شود ولی برای ضرب و تقسیم از آن جایی که از استاندارد و حل اعشار ثابت است الگوریتم پیچیده تر از ممیز شناور می شود

$$\begin{array}{r} 1.0011 \times 2^{1110} \\ +\ 1.1010 \times 2^{0001} \\ \hline \end{array}$$

نمایش اعداد BCD: (Binary Coded Decimal)

در این نمایش هر رقم دهدهی را در 4 بیت نمایش می دهیم. این نوع نمایش به خاطر استفاده کاربران از کامپیوتر ایجاد شده. در واقع کامپیوتر ورودی را به صورت BCD دریافت می کند و سپس به باینری تبدیل می کنیم

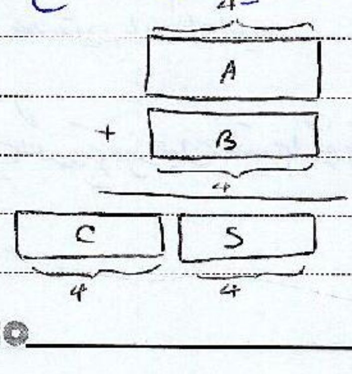
$$\begin{array}{r} 595 \\ +\ 683 \\ \hline \end{array}$$

	4	4	4
	0101	1001	0101
+	0110	1000	0011

یکی از روش های انجام عملیات در BCD این است که ورودی BCD را به باینری تبدیل کرده و سپس عملیات را انجام داده و دوباره به BCD تبدیل می کنیم ولی این الگوریتم بسیار پیچیده است در نتیجه از راه حل های زیر استفاده می کنیم:

I) BCD Adder:

در جمع باینری دو عدد n بیتی یک بیت بیشتر از ورودی ها بود ولی در این روش حاصل جمع دو عدد 4 بیتی برابر 8 بیت می شود یکی برای یکان و یکی برای دهگان



Subject:
Year. Month. Date. ( )

4 بیت	
ورودی دهدهی	$\overline{AB}$: X

A	B	: Y

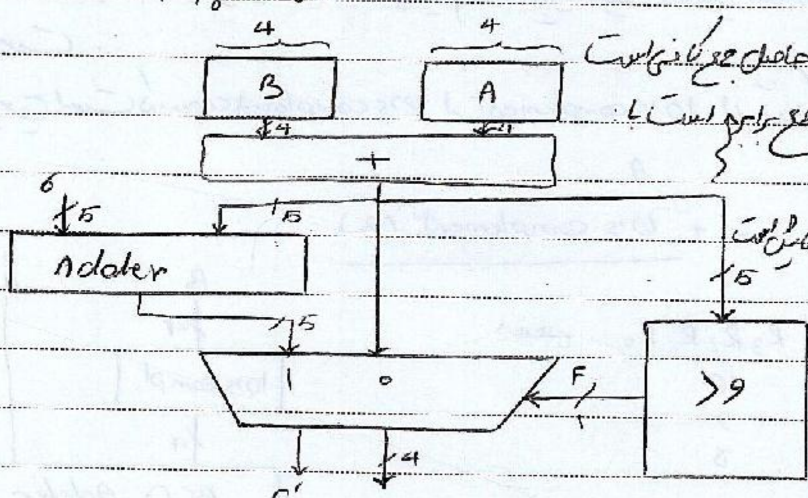
$$\begin{cases} Y = A * 16 + B \\ X = 10 \cdot A + B \end{cases}$$

$$Y - X = 6A \implies Y = 6A + X$$

در نتیجه اگر عدد بزرگتر از ۹ شود باید عدد ۶ را با آن جمع شود. زیرا همان ۶A برابر یک می‌شود در
رابطه بالا A=1 است در نتیجه $Y = 6 + X$

select در MUX F	A+B
0	0
0	1
0	2
.	.
0	9
1	10 01010 $\rightsquigarrow \bar{C} S_3 \bar{S_2} S_1 \bar{S_0}$
1	11 01011
1	12
1 (+6)	13
1	14
1	15
1	16
1	17
1	18 10010

$$F = C + S_3 S_2 + S_3 S_1$$



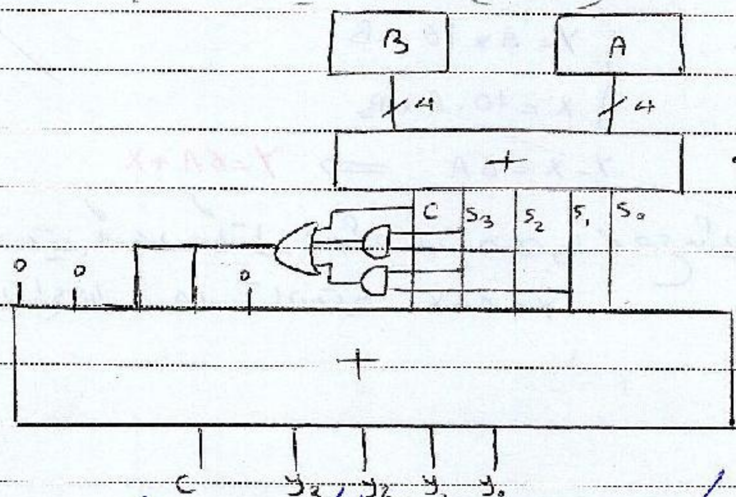
5 بیت برای نمایش حاصل جمع کافی است
زیرا بزرگترین حاصل جمع برابر است با
$18 + 6 = 24$
5 بیت برای نمایش کافی است

دهدهی
نهایتاً برابر یک است پس با یک بیت می‌توان
مشخص کرد.

PAPCO

Subject:
Year. Month. Date. ( )

؟ توی اول الگوریتم هم که جمع را انجام میدهیم چون دو رقم جمع داریم؟



این مدار مربوط به جمع دو عدد BCD یک رقمی است که برای BCD های چند رقمی کافی است خروجی ها را cascade کنیم.

الگوریتم اول از یک MUX استفاده می‌کند ولی در مدار بالا در هر مرحله یک جمع اضافه انجام می‌دهیم (دو جمع انجام می‌دهیم) سرعت الگوریتم اول بیشتر است.

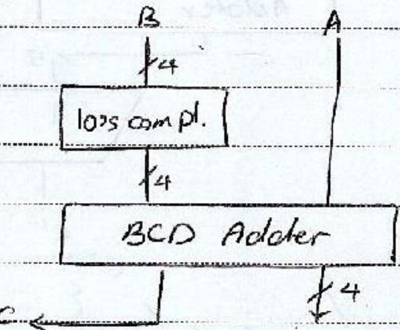
II) تفریق BCD

یکی از ساده‌ترین روش‌ها برای تفریق دو عدد BCD این است که هر عدد را به باینری تبدیل کرده سپس 2's complement کرده و خروجی را محاسبه می‌کنیم که تبدیل یک عدد باینری 2's complement شده به BCD کار دشواری است.

راه دیگر این است که به جای 2's complement از 10's complement استفاده کنیم.

$$A - B \rightsquigarrow A + 10\text{'s complement}(B)$$

$B_3 B_2 B_1 B_0$	$R_3 R_2 R_1 R_0$ (دهدهی)
0	10
1	9
2	8
⋮	⋮
9	1



در اینجا برای R_3 تا R_0 هر کدام یک جدول کارنو کشیده و روابط را بدست می‌آوریم.

PAPCO

Subject:

Year. Month. Date. ( )

طراحی Control Unit &

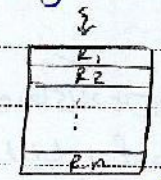
سیم بندی شده : hard wired

برنامه نویسی شده : micro programmed

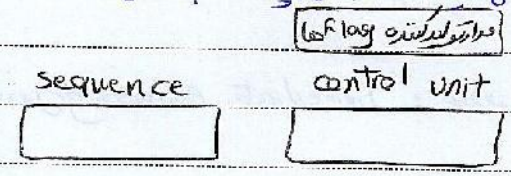
برای طراحی CPU داخل زیر را باید انجام دهیم:

۱- طراحی data path : به طراحی اجزای داخل پردازنده که برای نگهداری اطلاعات، انتقال یا انتخاب اطلاعات به کار می‌روند می‌گوییم. MUX، register، reg File، BUS، ALU

(همان اجزایی که برروی داده‌ها کار می‌کنند data path محسوب می‌شوند.)



۲- طراحی control unit (واحد کنترل) : به طراحی اجزای داخل پردازنده که کنترل دریافت داده و ارسال را بر عهده دارند برای اجرای توالی عملیات در داخل پردازنده کار می‌رود.



مجموعه‌ای از دستورهای قابل قبول برای پردازنده : (Instruction Set Architecture ISA

به صورت استاندارد بیان می‌شود.

$\{I_1, I_2, \ldots, I_n\}$

از روی مجموعه دستورالعمل‌ها اطلاعات زیر باید استخراج شود:

۱- CISC ، RISC

۲- تعداد دستورالعمل‌ها

۳- نوع دستورالعمل

۴- شیوه‌های آدرس دهی عملوندها

۵- تعداد عملوندها

PqPCO

Subject:
Year.　Month.　Date.　(　)

دستورالعمل‌ها از نظر تعداد عملوندها به دسته‌های زیر تقسیم می‌شوند:

1) ۰ آدرسی: نیاز به هیچ عملوندی ندارد. معمولاً ماشین پشته‌ای که با stack کار می‌کنند و داده‌ها درون stack قرار دارد.

2) ۱ آدرسی: فقط یک عملوند درون دستورالعمل مشخص می‌شود.

3) 2 آدرسی: ۲ تا operand دارند. ADD BX عدد دوم درون یک ثبات به نام Accumulator قرار می‌گیرد.

⋮

n آدرسی: n تا operand دارد.

در ماشین‌های CISC می‌توانند همهٔ انواع این دستورالعمل‌ها وجود داشته باشد ولی برای سادگی و در ارائه CPU از ماشین‌های RISC استفاده می‌کنیم که تعداد operand هایی که می‌گیرد مشخص و محدود است.

در شیوه‌های آدرس‌دهی باید مشخص شود که operand های موجود آدرس داده را مشخص می‌کنند یا خود داده است. انواع شیوه‌های آدرس‌دهی به صورت زیر است:

۱- آدرس‌دهی ضمنی Implicit Addressing: دستوراتی که نیاز به operand ندارند

CLC , STD , PUSHF

۲- آدرس‌دهی بلافصل Immediate Addressing: عدد مورد نظر مستقیماً به عنوان operand داده می‌شود.

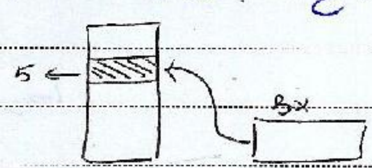
ADD 5

۳- آدرس‌دهی ثباتی مستقیم Direct Register Addressing: داده‌ها درون یک ثبات قرار می‌گیرد و شمارهٔ ثبات به عنوان operand در مقابل دستور قرار می‌گیرد.

ADD BX

۴- آدرس‌دهی ثبات غیرمستقیم Indirect Register Addressing: آدرس دادهٔ مورد نظر درون یک ثبات وجود دارد. در واقع داده به صورت pointer قابل دسترسی است

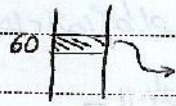
ADD [BX]



PAPCO

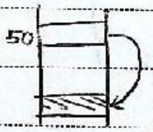
Subject:
Year. Month. Date. ( )

۵ـ آدرس دهی مستقیم (Direct Addressing) : به صورت مستقیم از روی حافظه اصلی داده خوانده می‌شود.



ADD [60]

۶ـ آدرس دهی غیرمستقیم (Indirect Addressing) : همانند حالت 4 است با این تفاوت که آدرس داده مورد نظر درون خانه‌ای از حافظه است. در مقایسه این دو روش حالت 4 بهتر است زیرا دسترسی به رجیستر ساده‌تر از حافظه اصلی است.



ADD [[50]]

۷ـ آدرس دهی شاخص دار (Index Addressing) : برای نوشتن حلقه‌های loop به کار می‌رود و از طریق index های مختلف می‌توان به خانه‌های بعدی آدرس مورد نظر دسترسی داشته باشیم.

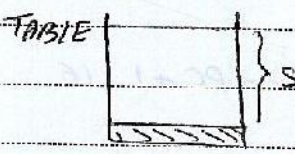
ADD TABLE[SI]

آدرس شروع ثابت و index تغییر می‌کند

۸ـ آدرس دهی یا پایه (Base Addressing) : در این حالت آدرس شروع آرایه تغییر می‌کند و index ثابت است که یک رجیستر به نگه داشتن آدرس شروع در حافظه اصلی اختصاص می‌دهند.

ADD [BX + 4]

۹ـ آدرس دهی شاخص دار با پایه : هم آدرس شروع آرایه و هم اندیس آن قابل تغییر است.



ADD BX[SI]

۱۰ـ آدرس دهی خودافزایشی یا خودکاهشی (Auto Inclement/Declement) : دسترسی که به صورت اتوماتیک دارد مقدار یک بیت را زیاد یا کم می‌کند

MOVSB , Loop

اندیس SI یا DI را انجام می‌دهد ← | مقدار CX زیاد می‌شود ←

PAPCO

Subject:
Year. Month. Date. ( )

در پردازنده های RISC تمام شیوه های آدرس دهی 3، 4، 5 و 6 انجام می شود.

مراحل انجام طراحی یک CPU به صورت زیر است:

1) Data path: تعداد ثبات های مورد نیاز، تعداد ALU، نحوه اتصالات، BUS ها مشخص می شود

ISA

2) Control Unit: sequence، طراحی مدار کنترلی، پیاده سازی الگوریتم فون نیومن

الگوریتم فون نیومن:

همان طور که اجرای دستورات به صورت ترتیبی انجام می شود، اجرای یک دستور نیز نیاز به مراحل ترتیبی دارد که به صورت زیر است:

1) خواندن دستورالعمل Instr. Fetch (واکشی)

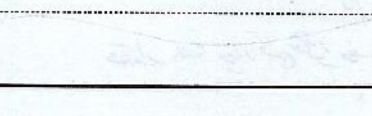
2) بازشناسی دستورالعمل Instr. Decode:

3) خواندن عملوندها (operand)

4) انجام عملیات (اجرای دستور) Execute

5) نوشتن نتیجه در مقصد دستورالعمل (write back)

program counter: PC



6) PC = PC + 1

7) برو به 1

PAPCO

Subject:

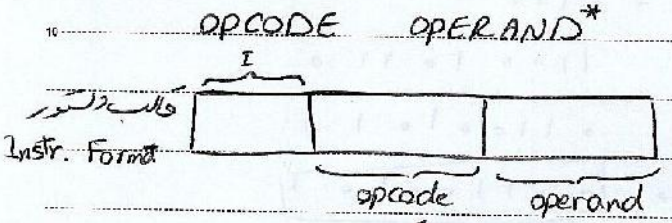
Year. Month. Date. ( )

هم سرعت اجرای الگوریتم فول نیومن بیشتر باشد سرعت اجرای دستور و در نتیجه سرعت اجرای الگوریتم بیشتر می‌شود.

برای بهبود الگوریتم فول نیومن از pipeline استفاده می‌شود که منتظر خارج شدن یک دستور از الگوریتم نمی‌شویم و در یک لحظه همهٔ مراحل الگوریتم در حال انجام کار است.

sequencer : یک شمارنده است که با تولید شماره مراحل اجرای الگوریتم اجرای یک دستور را مشخص می‌کند و با افزایش مقدار آن مراحل مختلف تولید و مشخص می‌شود. تعداد کدام مورد نیاز یک INS را مشخص می‌کند.

قالب یک دستور به صورت زیر است:



اگر تعداد کل دستورات موجود k باشد نیاز به $\lceil \log_2 k \rceil$ بیت دارد.

نکته: در پردازنده‌های RISC تعداد بیت مربوط به opcode و operand ثابت است ولی این تعداد در CISC می‌تواند متفاوت باشد.

در این روش تعدادی از opcode ها را به دستوراتی که با حافظه کار می‌کنند اختصاص می‌دهیم و تعدادی را به ثبات‌ها اختصاص می‌دهیم که در هر کدام یک بیت برای مشخص کردن direct و Indirect بودن اختصاص می‌دهیم.

Subject:
Year. Month. Date. ( )

مثال: ضرب زیر را به صورت بوث انجام دهید:

```
  1010  --(متمم ۲)-->  0110
× 1101  --(متمم ۲)-->  0011
```

```
     0000   1101   [0]
   + 0110
   ----------------
     0110   1101   0
→    0011   0110   1
   + 1010
   ----------------
     1101   0110   1
→    0110   1011   0
   + 0110
   ----------------
     1100   1011   0
→    0110   0101   1
→  [ 0011   0010   1   1 ]
   + 1010
   ----------------
     1101   0010   1
→    0110   1001   0
   + 0110
   ----------------
     1100   1001   0
→    0110   0100   1
```

تمرین:

```
  11000
× 01010
```

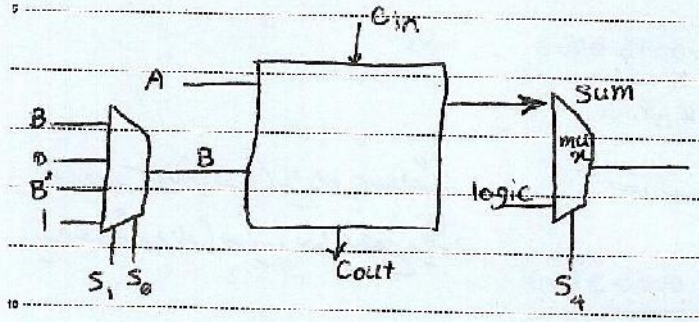
```
  16 8 4 2 1
  01010        10
× 11000        -8    -80
               24
               240
  ----------
  01010000
  010100000          01010
  ----------
  011110000
     16
                16
                32
```

Subject.

Year. Month. Date. ( ).

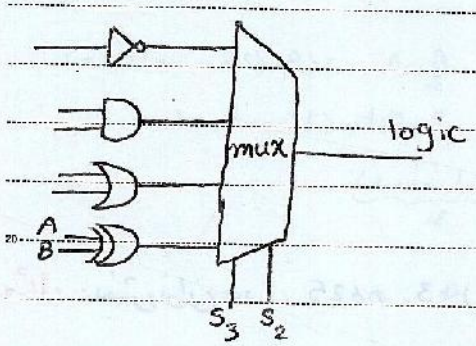
مثال: یک ALU ساده تنها با یک Full Adder بهست آورید.

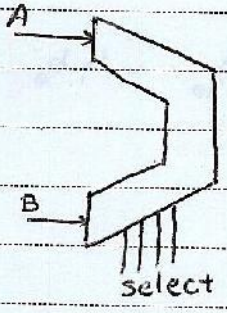
یکی از راه‌ها این است که روابط را برای 1 بیت طراحی کنیم و سپس همین مدار را بهست سرهم برای تعداد بیش‌تر به کار می‌بریم. به این روش bit slice می‌گویند.



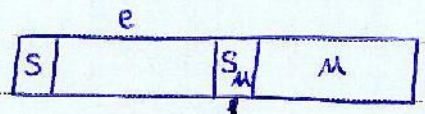
S_1	S_0	Cin	Function
0	0	0	A+B
0	0	1	A+B+1
0	1	0	A
0	1	1	A+1
1	0	0	A+B = A+B-1
1	0	1	A+$\bar{B}$+1 = A-B
1	1	0	A-1
1	1	1	A

logic:

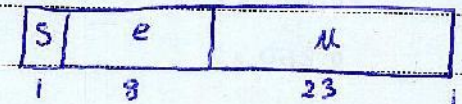




مثال: مقوان Floating point را به صورت زیر نمایش داد.



می‌توان به صورت مکمل دو یا اندازه علامت نمایش داد.



عدد bias که باید به آن اضافه شود 127 است.

Subject:
Year. Month. Date. ( )

آیا این را هم به مبنای 16 باید نرمال کنیم

مثال: با فرض نمایش اعداد به صورت ممیز شناور نمایش عدد زیر را به دست آورید.

$0.125 \times 16^5 = (-1)^S M \times B^{e-biased}$

S	e	M
1	7	24

مبنای 16 0.20×16^5 $0.0010\ 0000 \ldots 00$

در مبنای 16 است پس باید 4 بیت 4 بیت در نظر بگیریم چون 4 بیت صفر نیست پس همچنین نرمال است.

نرمال در حالت عادی فقط اولین عدد اعشار غیر صفر باشد در 4 بیت دوتا صحیح می‌شود

0.00000100

است باید نرمال بشود

مثال: مقدار کوچک‌ترین و بزرگ‌ترین عدد قابل نمایش را به دست آورید.

$2^S\left(\frac{1}{2} - b_{31}\right)\left(1 + \sum_{i=0}^{23} 2^{i-24} b_i\right)$

$S = -64 + \sum_{i=24}^{30} 2^{i-24} b_i$ کوچک‌ترین عدد $\underbrace{100 \ldots 00}_{31}$

$b_{31} b_{30} \ldots b_1 b_0$

بزرگ‌ترین عدد $B_{31} = 0 \quad 0 < B_i < 23 \rightarrow B_i = 1$

$S \quad 24 < b_i < 30 \quad B_i = 1$

$\underbrace{0111 \ldots 11}_{31}$

مثال: بخش توان عدد 143.0625 در استاندارد IEEE چگونه نمایش داده می‌شود (single)

143 : 10001111

$0.0625 \times 2 \rightarrow .0001$ نرمال $\rightarrow 1.000111100001$

0.125×2

0.250×2

0.500×2

1.0

PAPCO